UNIVERZITET SINGIDUNUM

Saša Adamović

# ZAŠTITA INFORMACIONIH SISTEMA

## JAVA IMPLEMENTACIJA KRIPTOGRAFSKIH MEHANIZAMA




Beograd, 2015.

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

**UNIVERZITET SINGIDUNUM**

**FAKULTET ZA INFORMATIKU I RAČUNARSTVO**

Saša Adamović

# Zaštita informacionih sistema

Prvo izdanje

Beograd, 2015.

**Zaštita informacionih sistema**

*Autor:*
dr Saša Adamović

*Recenzenti:*
dr Milan Milosavljević
dr Mladen Veinović
dr Aleksandar Jevremović

*Izdavač:*
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32

*Za izdavača:*
Prof. dr Milovan Stanišić

*Tehnička obrada:*
Saša Adamović

*Dizajn korica:*
Aleksandar Mihajlović

*Godina izdanja:*
2015.

*Tiraž:*
400 primeraka

*Štampa:*
Mobid, Loznica

ISBN: 978-86-7912-589-7

*„Ne postoji razlika između teorije i prakse u teoriji, ali u praksi je ima."*

Jan L. A. van de Snepscheut

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

## SADRŽAJ


1. UVOD U JAVA KRIPTOGRAFSKU ARHITEKTURU ..... 17

**1.1 *JCA* ekstenzija ..... 18**

**1.2 *JCE* ekstenzija ..... 19**

**1.3 Arhitektura *JCE* ekstenzije ..... 20**

**1.4 Drugi bezbednosni serivsi u Javi ..... 25**

**1.5 Dinamička registracija provajdera ..... 26**

2. KRIPTOLOŠKI KLJUČEVI I IMPLEMENTACIJA SIMETRIČNIH ŠIFRI ..... 33

**2.1 Generisanje slučajnih brojeva ..... 35**

**2.2 Generisanje kriptografskih pseudo slučajnih brojeva ..... 36**

2.2.1 Generisanje slučajne vrednosti ..... 37

**2.3 Generisanje kriptološkog ključa ..... 40**

2.3.1 Nezavisno inicijalizovani algoritmi ..... 42

2.3.2 Specifično inicijalizovani algoritmi ..... 43

2.3.3 Generisanje simetričnog ključa preko SecretKey klase ..... 43

**2.4 Transparentnost generisanih ključeva ..... 45**

2.4.1 Pretvaranje ključa po specifikaciji – način 1 ..... 46

2.4.2 Pretvaranje ključa po specifikaciji – način 2 ..... 52

**2.5 Kategorizacija simetričnih šifri ..... 53**

2.5.1 Upravljanje ključevima ..... 54

**2.6 Simetrične šifre - šifarska klasa ..... 55**

2.6.1 Inicijalizovanje simetrične šifre..... 58

2.6.2 Šifrovanje blokovskom šifrom ..... 60

2.6.3 Šifrovanje u *CFB* režimu – blokovska šifra ..... 64

2.6.4 Dešifrovanje u *CFB* režimu – blokovksa šifra ..... 67

**2.7 Šifrovanje na osnovu lozinke - *PBE* ..... 69**

2.7.1 Šifrovanje na osnovu lozinke – blokovska šifra ..... 70

**2.8 Šifrovanje sa sekvencijalnom šifrom *RC4* ..... 73**

2.8.1 Implementacija *RC4 šifre* ..... 75

3. ASIMETRIČNI KLJUČEVI I IMPLEMENTACIJA ASIMETRIČNIH ŠIFRI..... 77

**3.1 Generisanje asimetričnih ključeva ..... 78**

3.1.1 Generisanje ključeva nezavisno od algoritma ..... 80

3.1.2 Kodne šeme za kodovanje ključeva ..... 82

3.1.3 Zapisivanje i učitavanje asimetričnih ključeva ..... 84

**3.2 Asimetrične šifre – šifarska klasa ..... 86**

3.2.1 Šifrovanje sa javnim ključem - *RSA* ..... 88

3.2.2 Dešifrovanje sa privatnim ključem -RSA ..... 89

**3.3 Protokoli za razmenu tajnog ključa ..... 90**

3.3.1 Difi-Helman protokol ..... 91

4. HEŠ FUNKCIJE, INTEGRITET PORUKE I DIGITALNI POTPIS..... 97

**4.1 Kriptografske heš funkcije ..... 98**

4.1.1 *MD5* jednosmerna funkcija ..... 99

4.1.2 SHA jednosmerna funkcije ..... 99

**4.2 Heš funkcije – implementacija ..... 101**

**4.3 Integritet poruke – *MAC* ..... 106**

4.3.1 Generisanje *HMAC* vrednosti ..... 107

**4.3 Digitalni potpis ..... 110**

4.3.1 Digitalno potpisivanje ..... 111

4.3.2 Verifikovanje digitalnog potpisa ..... 113

4.3.3 Slabosti i alternative ..... 115

5. UPRAVLJANJE KLJUČEVIMA I SERTIFIKATIMA ..... 116

**5.1 Digitalni sertifikati ..... 116**

**5.2 Upravljanje sa Java skladištem ključeva ..... 119**

5.2.1 Generisanje sertifikata ..... 121

5.2.2 Skladišta za simetrične ključeve ..... 125

5.2.3 Učitavanje simetričnih ključeva iz skladišta ..... 127

**5.3 Upravljanje sertifikatima po X.509 standardu ..... 128**

5.3.1 Kreiranje zahteva za izdavanje sertifikata - *CSR* ..... 133

5.3.2 Lista povučenih sertifikata – *CRL* ..... 134

5.3.3 Šifrovanje sa digitalnim sertifikatima ..... 138

5.3.4 Dešifrovanje sa privatnim ključem ..... 139

5.3.5 Jedan scenario sa RSA i tajnim ključem ..... 140

6. *NIST* STATISTIČKA BATERIJA TESTOVA ..... 144

**6.1 Pregled statističkih testova ..... 144**

6.1.1 Ispitivanje učestalosti u nizu ..... 144
6.1.2 Ispitivanje učestalosti u bloku ..... 145
6.1.3 Ispitivanje uzastopnih ponavljanja istih bitova u nizu ..... 146
6.1.4 Ispitivanje najdužeg uzastopnog ponavljanja jedinica u bloku..... 147
6.1.5 Ispitvanje stanja binarne matrice ..... 149
6.1.6 Ispitivanje diskretne Furijerove transformacije..... 150
6.1.7 Ispitivanje nepreklapajućih uzoraka ..... 151
6.1.8 Ispitivanje preklapajućih uzoraka ..... 152
6.1.9 Maurerov univerzalni statistički test ..... 153
6.1.10 Ispitivanje linearne kompleksnosti ..... 154
6.1.11 Serijski test učestalosti ..... 155
6.1.12 Ispitivanje približne entropije ..... 157
6.1.13 Ispitivanje slučajnih zbirova..... 158
6.1.14 Ispitivanje slučajne digresije ..... 159
6.1.15 Ispitivanje slučajne promenljve digresije..... 160

**6.2 Java aplikacija sa implementiranom baterijom testova..... 162**

7. JEDAN PRIMER SINTEZE SOPSTVENOG IZVORA SLUČAJNOSTI..... 165

**7.1 Uloga izvora slučajnosti u kriptografiji ..... 165**

**7.2 Generisanje slučajnosti na osnovu avio saobraćaja..... 166**

7.2.1 Arhitektura izvora slučajnosti ..... 167
7.2.2 Prikupljanje i obrada podataka..... 168
7.2.3 Analiza kvaliteta izvora slučajnosti ..... 169
7.2.4 Java program za prikupljanje podataka u realnom vremenu ..... 173

DODATAK - PROJEKTNI ZADACI I LABORATORIJSKE VEŽBE ..... 176

**8.1 Teme za projektni zadatak ..... 176**

**8.2 Laboratorijske vežbe ..... 178**

LITERATURA ..... 185

REČNIK POJMOVA ..... 187

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

## PREDGOVOR

Zaštita informacionih sistema je udžbenik namenjen studentima Fakulteta za informatiku i računarstvo na Univerzitetu Singidunum. Predviđen je za pripremu ispita iz predmeta „Praktikum – Zaštita informacionih sistema“. Takođe, udžbenik mogu koristiti svi oni kojima je potrebno praktično znanje za pravilnu implementaciju kriptografskih mehanizama u programskom jeziku Java, za uspešan razvoj savremenih informaciono bezbednih poslovnih sistema. Udžbeniku „Zaštita informacionih sistema“, prethodili su udžbenici „Kriptologija 1“, „Kriptologija 2“ i „Osnovi teorije informacija i kodovanja “. Navedeni udžbenici predstavljaju teorijski i praktični fundament koji je neophodan za razumevanje i praktično savladavanje velikog broja primera u ovom udžbeniku. Tokom pisanja udžbenika važila je pretpostavka da budući čitaoci već imaju dobro teorijsko znanje iz kriptologije i teorije informacija. Verujemo da svi oni koji čitaju ovaj udžbenik poznaju terminologiju: provera identiteta, javni ključ, privatni ključ, digitalni sertifikat, tajni ključ, šifrovanje, dešifrovanje i terminologiju iz oblasti teorije informacija.

Sa duge strane, ovaj udžbenik predstavlja odličnu vežbu ili obuku za Java programere, koji će upoznati novu terminologiju iz Java kriptografske arhitekture, pravila i principe za pravilnu implementaciju kriptografskih klasa u poslovne Java aplikacije i servise. Iz tog razloga ovaj udžbenik može da se koristi i kao uputstvo za upotrebu Java kriptografske arhitekture - *JCA*. Zbog svoje velike praktične vrednosti, olakšaće studentima rešavanje problema koji se odnose prvenstveno na pravilnu implementaciju kriptografskog Java koda, bez obzira da li je reč o velikim poznavaocima Java programskog jezika.

Razvoj savremenih informacionih tehnologija i edukativnih materijala za njih doveo je do premeštanja različitih vrednosti na Internet mrežu i njene servise. Osnovno pitanje koje se izdiže danas, jeste kako te vrednosti zaštititi od zlonamernih akcija, odnosno kako iškolovati kadar sposoban da se uhvati u koštac sa savremenim bezbednosnim rizicima i izazovima.

Jedan od osnovnih problema pri učenju kriptologije predstavlja njena kompleksnost i zasnovanost na složenim matematičkim principima i formulama. Najveći broj bezbednosnih rešenja danas ne zahteva detaljno poznavanje pomenutih matematičkih principa već samo njihovo osnovno razumevanje i povezivanje sa ulogom u praktičnim rešenjima. Prebacivanjem fokusa iz domena matematičkih osnova bezbednosnih rešenja u domen praktične primene kod savremenih informacionih i telekomunikacionih tehnologija značajno se menja i profil studenata u ovoj oblasti. Iz tog razloga je ovo preusmerenje neophodno podržati odgovarajućim edukativnim pristupima, materijalima, savremenim metodama i uslovima za učenje.

Na osnovu dosadašnjeg iskustva u ovoj oblasti, većina ljudi pojednostavi ulogu i sam značaj kriptografije. Često možemo da čujemo komentar „mi smo dovoljno bezbedni, šifrovali smo podatke“ ili „izračunali smo heš od lozinke“, ali mišljenje mojih kolega i moje mišljenje je da to praktično ništa ne znači. Da bismo se uverili koliko je procena snage jednog kriptografskog sistema kompleksna, postavićemo sledeća pitanja na koja je potrebno odgovoriti preko poglavlja sadržanih u udžbeniku. Koliko je kriptološki ključ jak? Šta predstavlja efikasnu dužinu kriptološkog ključa? Koji režim šifrovanja ste koristili kod simetričnih šifara? Na koji način upravljate ključevima? Kako generišete ključeve? Šta je uzeto za izvor slučajnosti? Da li ste testirali informacionu snagu ključa? Šta znači sertifikat od poverenja? Da li pravite dobar odabir šifre? Da li su podaci šifrovani na memorijskom modulu ili se nalaze u šifrovanoj formi za vreme transmisije? Na osnovu postavljenih pitanja, ukazujemo na gradivne komponente jednog kriptografskog sistema preko koga možemo da postavimo teorijske okvire koje treba uzeti u razmatranje pri proceni snage kriptografskog sistema. U protivnom, svi drugi komentari poput onih na početku pasusa predstavljaju kriptografsku neistinu koja vrlo lako može dovesti do ozbiljnih posledica zbog neosnovanog osećaja za stepen implementirane bezbednosti.

Kriptografija kao oblast uključuje šifrovanje sa simetričnim i asimetričnim algoritmima, jednosmerne funkcije i različite digitalne sertifikate za digitalno potpisivanje i šifrovanje. Kada je reč o šifrovanju, bitno je naglasiti da taj deo kriptografije posmatramo sa aspekta perfektnih i neperfektnih šifara. Novinu u kriptografiji predstavljaju novi trendovi razvoja ove oblasti koja se okreće ka smeru perfektnih bezbednosnih sistema na fizičkom sloju. Kada je reč o autentifikaciji,

autentifikacija je direktno povezana sa kriptografijom i tu vezu potvrđuje primena kriptografije sa javnim ključem u protokolima za rešavanje problema jednostrane ili uzajamne autentifikacija. Upravo, cilj ovog udžbenika je da preko sledećih poglavlja edukuje novu generaciju kriptografa koja će moći da odgovori svim izazovima u budućnosti.

U prvom poglavlju, napravili smo detaljan uvid u Java kriptografsku arhitekturu. Diskutovali smo o *JCA* i *JCE* kriptografskoj ekstenziji i različitim provajderima bezbednosti koji obezbeđuju široku paletu kriptografske funkcionalnosti. Takođe, prikazali smo specifikaciju dostupnih algoritama i određena ograničenja za Java *SunJCE* provajder. Na kraju poglavlja, napravljen je osnovni pregled i drugih bezbednosnih nekriptografskih servisa u Javi.

U drugom poglavlju bavimo se implementacijom simetričnih šifri. Polazeći od osnovnih Kerkehofih principa (tajnost leži u snazi ključa), prvo je pokazana implementacija algoritma za generisanje „kriptografski“ slučajnih brojeva. Zatim, prešli smo na upotrebu klasa za kreiranje kriptografske specifikacije ključa na različite načine (transparentno ili netransparentno). Posle razumevanja procesa za generisanje tajnog ključa u Javi, implementirali smo nekoliko simetričnih šifri sa različitim parametrima algoritama, pod čim podrazumevamo naziv šifre, šifarski režim i šemu dopune otvorenog teksta. Pored implementacije simetričnih šifri (blokovske i sekvencijalne), razmatrali smo metode za generisanje kriptološkog ključa na osnovu korisničkih lozinki – *PBE* algoritam.

U trećem poglavlju bavimo se implementacijom asimetričnih šifri, kao i generisanjem para ključeva (javni i privatni). Posebno stavljamo fokus na kodne šeme (*X.509*, *PKCS#8*) koje se koriste za javni i privatni ključ. Zatim, implementiramo šifre iz asimetrične kriptografije za operacije šifrovanja ili dešifrovanja preko *RSA* algoritma. Na kraju ovog poglavlja razmatramo o protokolima za razmenu simetričnih ključeva koji kao osnovu koriste asimetričnu kriptografiju. Pored teorijske osnove, data je kompletna implementacija algoritma za generisanje Difi-Helman parametara i algoritma za razmenu ključa preko Difi-Helman protokola (Alisa i Bob scenario), preko koga Alisa i Bob razmenjuju šifrovane poruke.

U četvrtom poglavlju, bavimo se implementacijom jednosmernih funkcija, jednosmernih *HMAC* funkcija za proveru integriteta poruke (u prenosu preko kanala) i digitalnim potpisom. Pored implementacije *MD5* i *SHA* verzija jednosmernih funkcija, analiziramo performanse istih funkcija. Zatim, diskutujemo o dizajnu *HMAC* algoritma i njegovoj pravilnoj implementaciji preko istih Java klasa koje se koriste za generisanje tajnog ključa za simetrične šifre. Prikazaćemo dva načina za generisanje tajnog ključa. Na kraju poglavlja obavljena je pravilna implementacija generisanja digitalnog potpisa sa javnim privatnim ključem, kao i implementacija Java koda za verifikovanje digitalnog potpisa javnim ključem.

Peto poglavlje udžbenika odnosi se na upravljanje sa tajnim ključevima i sertifikatima. Polazeći od teorijske osnove digitalnih sertifikata, prelazimo na metode za generisanje digitalnih sertifikata preko alata *KeyTool* u Javi. Upoznaćemo se sa svim funkcionalnostima ovog alata (generisanje sertifikata, izvoz sertifikata, uvoz sertifikata, potpisivanje sertifikata i brisanje). Posle par primera u *KeyTool*-u, iste primere ćemo da realizujemo programski, tj. nezavisno kroz programski kod Java jezika. Osim, rada sa digitalnim sertifikatima, na sličan način ćemo generisati tajne ključeve koje ćemo takođe, uskladištiti preko Java *KeyStore*-a. Videćemo na koji način se reguliše kontrola pristupa skladišta, kako se određuju tipovi skladišta (*JKS*, *JCEKS*, *BCKS*, i td.) prema nameni, kao i kontrola pristupa entitetima unutar jednog skladišta (postavljanje lozinke za PBE šifrovanje skladišta). Na kraju petog poglavlja, bavimo se mehanizmima za proveru autentičnosti potpisanih sertifikata, mehanizmima za povlačenje sertifikata - *CRL* liste i demonstriramo jedan primere sa upotrebom digitalnog sertifikata za šifrovanje i dešifrovanje.

U šestom poglavlju dat je detaljan pregled *NIST* statističke baterije testova sa primerima. Osnovna namena ovih testova je da procene kriptološki kvalitet sekvenci dobijenih preko nekog izvora slučajnosti, a na osnovu kojih se kasnije generišu kriptoloških ključevi za zadate algoritamske specifikacije. Na kraju poglavlja, predstavljena je upotreba jedne grafičke aplikacija (razvijene na Univerzitetu Singidunum) u kojoj su implementirani skoro svi testovi za procenu stepena slučajnosti. Pored testova, aplikacija ima mogućnost da generiše slučajne vrednosti preko *AES*-a i *DES*-a.

U sedmom poglavlju, diskutovaćemo o jednom alternativnom izvoru slučajnosti, koji predstavlja naučni doprinos profesora i studenata sa Univerziteta Singidunum. Uloga ovakvih izvora je da se izgradi pouzdan generator kriptoloških ključeva preko koga će studenti imati mogućnost da u svojim kriptografskim programima uvedu sopstveni izvor slučajnosti za potrebe kriptografskih algoritama, za zamenu umesto do sada korišćene `SecureRandom` ili `Random` klase u Java programskom jeziku. Takođe, uz pomoć sopstvenog izvora slučajnosti, studenti imaju mogućnost da u svojim praktičnim radovima dizajniraju sisteme za distribuciju kriptoloških ključeva preko javnih komunikacionih kanala.

U poslednjem osmom poglavlju, predstavljene su preliminarne teme koje studenti mogu da izaberu i zadaci za laboratorijske vežbe koje su obavezne za proces savladavanja gradiva iz ovog udžbenika. Svaka uspešna realizacija projektnog rada vodi ka izradi kvalitetnog diplomskog rada, što pored stečenog praktičnog znanja iz oblasti zaštite informacija, predstavlja još jedan dodatni cilj.

Na kraju, zahvaljujem se recenzentima na korisnim sugestijama koje su unete u udžbenik. Posebnu zahvalnost dugujem profesorima, prof. dr Milanu Milosavljeviću i prof. dr Mladenu Veinoviću, koji su mi svojim savetima na osnovu dugogodišnjeg iskustva u ovoj naučnoj oblasti, neizmerno pomogli i bili inspiracija u pisanju udžbenika. Ovo je prvo izdanje, te su sve kritike i sugestije čitaoca dobro došle.

Beograd, 2015. godine

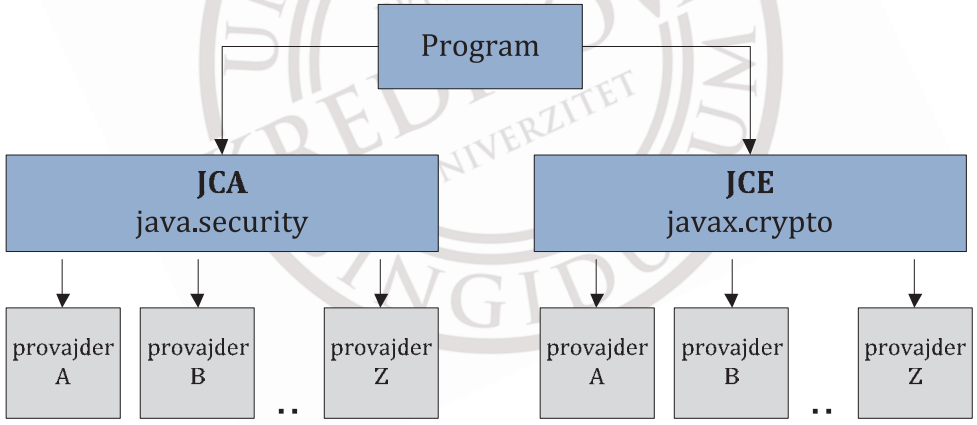
Saša Adamović

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

# 1. UVOD U JAVA KRIPTOGRAFSKU ARHITEKTURU

Izvoz kriptografije iz Sjedinjenih Američkih Država (engl. *U.S. Export Laws*), podrazumevao je transfer kriptografskih uređaja i tehnologije u druge zemlje van SAD-a. Do 1992. godine postojala su vrlo stroga ograničenja koja su se ticala izvoza u druge zemlje, ali posle 2000-te godine izvoz je dopušten, ali i dalje su ostale neke restrikcije. Od 2009. godine, izvoz kriptografije iz *SAD* koja se ne koristi u vojne svrhe je pod kontrolom Ministarstva za trgovinu, Biroa za industriju i bezbednost. Restrikcije koje još postoje, posebno se odnose na neuređene zemlje i terorističke organizacije. Na primer, vojna oprema, uređaji i namenski kriptografski programi, kao i konsultantske kriptografske usluge i dalje zahtevaju izvoznu dozvolu.

Java programski jezik obezbeđuje paletu kriptografskih funkcionalnosti koristeći dva *API*-ja (engl. *APIs*), *JCA* i *JCE* ekstenziju. Preko ove dve kriptografske biblioteke obezbeđene su sve kriptografske funkcije.




*Slika 1.1 Java kriptografski API-ji i provajderi funkcionalnosti*

Osnovni dizajnerski principi, uvedeni prilikom razvoja ovih kriptografskih ekstenzija su:

- Algoritamska nezavisnost;
- Algoritamska proširivost – mogućnost za uvođenje novih algoritama;

- Nezavisna implementacija – korišćenje provajdera za obezbeđenje različitih kriptografskih servisa;
- Interoperabilna implementacija – provajderi imaju mogućnost da međusobno sarađuju jedan sa drugim;

Sledi lista provajdera dostupna za *JCA* i *JCE* funkcionalnosti za Java platformu 7 (*JPSE7*):

- *SunPKCS11*
- *SUN*
- *SunRsaSign*
- *SunJSSE*
- *SunJCE*
- *SunJGSS*
- *SunSASL*
- *XMLDSig*
- *SunPCSC*
- *SunMSCAPI*
- *SunEC*
- *OracleUcrypto*

## 1.1 *JCA* EKSTENZIJA

**Java kriptografska arhitektura** – *JCA* (engl. *Java Cryptography Architecture*) predstavlja bezbednosno okruženje integrisano sa jezgrom Java aplikacije. Obezbeđuje sledeće klase i funkcionalnosti:

- `MessageDigest` - heš vrednosti,
- `Signature` - digitalni potpis,
- `KeyPairGenerator` - generisanje para ključeva (*PKI*),
- `KeyFactory` i `KeyStore` – upravljanje i čuvanje ključeva,
- `SecureRandom` – kriptografski izvor slučajnosti,
- `AlgorithmParameters` – upravljanje sa svim parametrima algoritma,

- `AlgorithmParameterGenerator` – generisanje svih parametara za traženi algoritam,
- `CertificateFactory` – generisanje digitalnih sertifikata,
- `CertPathBuilder` – uspostavljanje lančane veze između sertifikata,
- `CertStore` - upravljanje sa digitalnim sertifikatima (izdavanje sertifikata, povlačenje, verifikacija i td);

Fundamentalne klase i interfejsi *JCA* proširenja su:

- `javax.crypto.Cipher` – kriptografski šifarski objekat koji se koristi za šifrovanje ili dešifrovanje;
- `javax.crypto.CipherStream` – klase koje prilagođavaju šifre za rad sa Java I/O (ulazni/izlazni) tokovima podataka;
- `javax.crypto.KeyGenerator` – koristi se za generisanje simetričnog kriptološkog ključa;
- `java.security.KeyFactory` – koristi se za konvertovanje ključa u zadati format koji je definisan od strane šifre koja će se koristiti, a sve na osnovu inicijalnog materijala za ključ, generisanog na slučajan način;
- `java.security.KeyPairGenerator` – koristi se za generisanje novog para asimetričnih ključeva;
- `javax.crypto.Mac` – koristi se za generisanje *MAC* vrednosti za neki tekst predstavljen kao niz bajtova;

Takođe, postoje i mnoge druge klase i interfejsi u paketima, ali ove klase predstavljaju ključne klase i interfejse. Od navedenih klasa, verovatno najosnovnija je `Cipher` klasa, jer svaki kriptografski servis podrazumeva neki vid šifrovanja i dešifrovanja.

## 1.2 *JCE* EKSTENZIJA

**Java kriptografska ekstenzija** *JCE* (engl. *Java Cryptography Extension*) obezbeđuje programske biblioteke za primenu kriptografskih funkcija u Java kodu. Tačnije, *JCE* nasleđuje *JCA* i proširuje ga sa većim brojem klasa i provajdera. Jedan od dodatnih

provajdera je *SUNJCE*, koji uključuje jednu ili više kriptografskih klasa. Ne zaboravimo da podela koja je nastala razdvajanjem *JCA* i *JCE* ekstenzije, ne predstavlja podelu prema tehničkim ograničenjima, već prema tadašnjim političkim situacijama.

*JCE* obezbeđuje sledeće klase i funkcionalnosti:

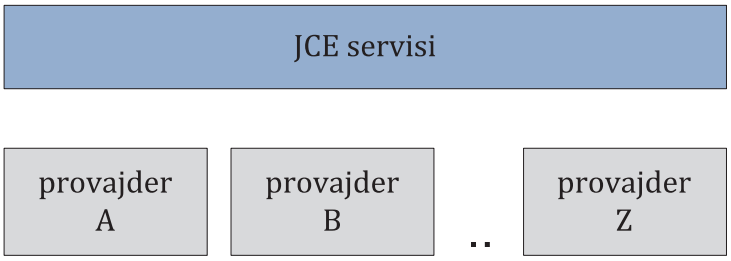
- `Cipher`- izvodi operaciju šifrovanja ili dešifrovanja,
- `KeyGenerator`- generiše kriptološke ključeve,
- `SecretKeyFactory`- izvršava se isključivo nad instancom `SecretKey`,
- `KeyAgreement` – razmena simetričnih ključeva između više strana,
- `Mac`– autentičnost poruke;

Možemo da kažemo da *JCA* i *JCE* zajedno predstavljaju kompletnu kriptografsku platformu. Međutim, postoje različite zamerke od strane programera. Neke od njih su lokacije kriptografskih klasa, da li je smeštena u `java.security` ili u `javax.crypto` paketu. Klase *JCA* ekstenzije su smeštene u `java.security` paketu, dok su klase *JCE* ekstenzije smeštene u `javax.crypto` paketu. Postoje dodatne zamerke i na hijerarhiju klasa. *JCA* se pridržava strogih obrazaca kada je reč o apstraktnim klasama, dok to nije slučaj sa *JCE*.

U narednom poglavlju razmotrićemo u kakvom su odnosu Java *JCE* ekstenzija i druge Java bezbednosne ekstenzije.

## 1.3 ARHITEKTURA *JCE* EKSTENZIJE

Treba imati na umu da je *JCE* dizajniran da obezbedi generički skup servisa koji je nezavisan od proizvođača softvera. Tada različiti proizvođači mogu napisati implementaciju tih servisa, koji se nazivaju provajderi. Različiti proizvođači nude provajdere sa različitim grupama funkcionalnosti i sa različitim karakteristikama.




*Slika 1.2* JCE *biblioteka i podrška od strane više provajdera*

Programeri prilikom razvoja nekog softvera mogu priključiti različite provajdere (više od jednog istovremeno A, B, ....Z) i da tako iskoriste mogućnosti i prednosti pojedinačno svakog od njih. Međutim, ovaj pristup u razvoja softvera sa *JCE* arhitekturom omogućava programerima da jednostavno zamene jedan provajder sa drugim.

Statički deklarisani provajderi nalaze se obično u fajlu na lokaciji: C:/Program Files/Java/jre7/lib/security/java.secutity. Sledi deo sadržaja fajla koji se odnosi na provajdere:

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunJCE
```

U nastavku ovog dela, prikazaćemo specifikaciju dostupnih algoritama i određene restrikcije provajdera *SunJCE.*

*Tabela 1.1 Dostupni algoritmi u provajderu* SunJCE

| **Klase** | **Naziv algoritma** |
|---|---|
| **AlgorithmParameterGenerator** | DiffieHellman |

| | |
|---|---|
| **AlgorithmParameters** | AES<br>Blowfish<br>DES<br>DESede<br>DiffieHellman<br>OAEP<br>PBEWithMD5AndDES<br>PBEWithMD5AndTripleDES<br>PBEWithSHA1AndDESede<br>PBEWithSHA1AndRC2_40<br>RC2 |
| **Cipher** | Sledeća tabela |
| **KeyAgreement** | DiffieHellman |
| **KeyFactory** | DiffieHellman |
| **KeyGenerator** | AES<br>ARCFOUR<br>Blowfish<br>DES<br>DESede<br>HmacMD5<br>HmacSHA1<br>HmacSHA256<br>HmacSHA384<br>HmacSHA512<br>RC2 |
| **KeyPairGenerator** | DiffieHellman |
| **KeyStore** | JCEKS |

| | |
|---|---|
| **Mac** | HmacMD5<br>HmacSHA1<br>HmacSHA256<br>HmacSHA384<br>HmacSHA512 |
| **SecretKeyFactory** | DES<br>DESede<br>PBEWithMD5AndDES<br>PBEWithMD5AndTripleDES<br>PBEWithSHA1AndDESede<br>PBEWithSHA1AndRC2_40<br>PBKDF2WithHmacSHA1 |

*Tabela 1.2 Dostupne šifre u provajderu* SunJCE

| Naziv algoritma | Režimi | Dopune |
|---|---|---|
| AES | ECB, CBC, PCBC, CTR, CTS, CFB, CFB8..CFB128, OFB, OFB8..OFB128 | NOPADDING, PKCS5PADDING, ISO10126PADDING |
| AESWrap | ECB | NOPADDING |
| ARCFOUR | ECB | NOPADDING |
| Blowfish, DES, DESede, RC2 | ECB, CBC, PCBC, CTR, CTS, CFB, CFB8..CFB64, OFB, OFB8..OFB64 | NOPADDING, PKCS5PADDING, ISO10126PADDING |
| DESedeWrap | CBC | NOPADDING |

| | | |
|---|---|---|
| PBEWithMD5AndDES, PBEWithMD5AndTripleDES1PBEWithSHA1AndDESede, PBEWithSHA1AndRC2_40 | CBC | PKCS5Padding |
| RSA | ECB | NOPADDING, PKCS1PADDING, OAEPWITHMD5ANDMGF1PADDING, OAEPWITHSHA1ANDMGF1PADDING, OAEPWITHSHA-1ANDMGF1PADDING, OAEPWITHSHA-256ANDMGF1PADDING, OAEPWITHSHA-384ANDMGF1PADDING, OAEPWITHSHA-512ANDMGF1PADDING |

*Tabela 1.3 Restrikcije veličine ključa* - KeyGenerator

| **Naziv algoritma** | **Standardna veličina ključa** | **Restrikcije** |
|---|---|---|
| **AES** | 128 | Veličina ključa može da bude 128, 192 i 256 bitova. |
| **ARCFOUR (RC4)** | 128 | Veličina ključa mora da bude u opsegu između 40 i 1024 bita. |

| | | |
|---|---|---|
| **Blowfish** | 128 | Veličina ključa mora biti deljiva sa 8, u opsegu od 32 do 448 bitova. |
| **DES** | 56 | Veličina ključ je tačno 56. |
| **DESede (Triple DES)** | 168 | Veličina ključa može da bude 112 ili 168 bitova. |
| **HmacMD5** | 512 | Nema restrikcije. |
| **HmacSHA1** | 512 | Nema restrikcije. |
| **HmacSHA256** | 256 | Nema restrikcije. |
| **HmacSHA384** | 384 | Nema restrikcije. |
| **HmacSHA512** | 512 | Nema restrikcije. |
| **RC2** | 128 | Veličina ključa mora da bude u opsegu između 40 i 1024 bita. |

## 1.4 DRUGI BEZBEDNOSNI SERIVSI U JAVI

Veći deo rada u Javi uložen je na bezbednost, mislimo prvo na obezbeđenje same Java virtuelne mašine – *JVM* (engl. *Java Virtual Machine*). Ovaj rad prvenstveno je usmeren na onemogućavanje zlonamernog programa ili loše logički osmišljenog koda da ugrozi rad računara ili aplikacije, kao i podatke korisnika. Sve to uključuje stvari kao što su menadžeri bezbednosti, tačnije servisi za privilegovane segmente koda, pravila koja kontrolišu šta je Java kodu dozvoljeno da uradi, a šta ne. Ovi servisi se nalaze u jezgru Java virtuelne mašine i upravljaju servisima bezbednosti unutar nje. Suštinski, stalno analiziraju kod koji se izvršava.

Međutim, kako se Java vremenom razvijala, postala prisutna i u širem poslovnom okruženju, javila se potreba za servisom autentičnosti i servisom za autorizaciju, da bi

se ovlastio veći broj korisnika u Java okruženju. Dobar primer su više korisnička okruženja. Više korisnička okruženja zahtevaju proveru identiteta korisnika, posebno u momentima kada treba da donesu autorizovanu odluku zasnovanu na identitetu. U cilju zamene nestandardnih rešenja od strane vlasnika Java koda, razvijen je Java servis za autentifikaciju i autorizaciju – *JASS* (engl. *Java Authentication and Authorization Service*).

Paralelno sa razvojem *JASS* servisa, razvijen je servis za bezbednu mrežnu komunikaciju – *JSSE* (engl. *Java Secure Sockets Extension*) i servis za bezbednosni programski interfejs *GSS-API* (engl. *Generic Security Services Application Programming Interface*) koji omogućava programeru da na bezbedan način promeni kriptografske mehanizme u aplikaciji ukoliko u budućnosti bude potrebe za tim, ali ne zahtevajući ponovno pisanje koda aplikacije. Ovi servisi primarno su usmereni na stvaranje bezbedne komunikacije i administrativne kontrole pristupa u Java programima.

*JCE* biblioteka dopunjava navedene bezbednosne servise (sa kriptografskog aspekta), pružajući podršku za obavljanje opštih kriptografskih aktivnosti u Javi, dok ostale tehnologije mogu ili ne, direktno koristiti *JCE* ekstenziju, jer su i one takođe, konceptualno izgrađene na istim kriptografskim tehnikama sa implementacijom po standardima koje propisuje *NIST* (engl. *National Institute of Standards and Technology*) organizacija. Sada možemo lako da pretpostavimo da je Java *JCE* ekstenzija u osnovi nastala kao podrška Java programerima, da na pravilan način, a to znači poštujući osnovna pravila i principe, implementiraju kriptografske mehanizme u svoje Java aplikacije.

## 1.5 DINAMIČKA REGISTRACIJA PROVAJDERA

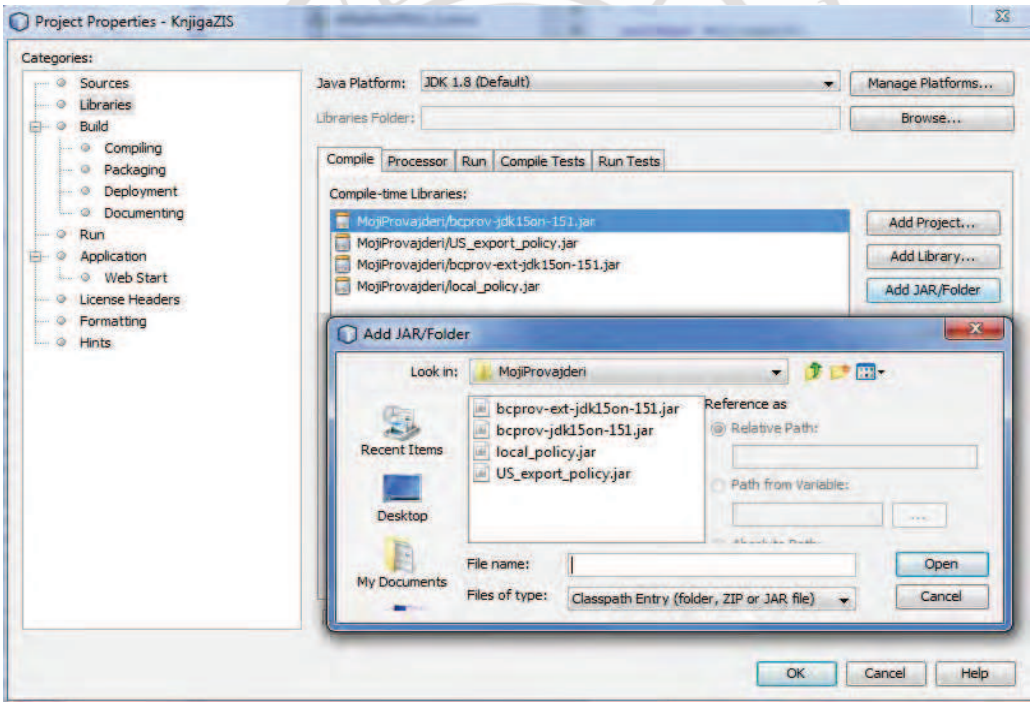
U ovom delu prikazaćemo jednu dinamičku registraciju, jednog od poznatijih provajdera. Reč je o kriptografskom provajderu *BouncyCastle. BouncyCastle* je kolekcija *API*-ja koji se koriste u kriptografske svrhe. Api ima podršku za Java i C# programski jezik.

*BouncyCastle* je australijskog porekla i iz tih razloga, kriptografske izvozna ograničenja *SAD*-a, ne odnose se na njega. Kompletnu *JAR* biblioteku moguće je preuzeti preko sledećeg linka:

**http://www.bouncycastle.org/latest_releases.html**

Posle preuzimanja biblioteke, potrebno je uraditi sledeće korake za uspešnu dinamičku registraciju *BounCycastle* provajdera.

1. Biblioteku *JAR* (bcprov-jdk15on-151.jar) sačuvati u glavnom projektu u folderu MojiProvajderi;
2. Preko interfejsa *NetBeans* razvojnog okruženja, vrlo jednostavno se uvezuje biblioteka sa kompletnim projektom;




*Slika 1.3 Instalacija drugih biblioteka*

U sledećem Java primeru, izvršićemo dinamičku registraciju *BouncyCastle* provajdera. Program ispisuje naziv i verziju raspoloživog provajdera.

```
import java.security.NoSuchAlgorithmException;
import java.security.Provider;
import java.security.Security;
import javax.crypto.NoSuchPaddingException;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class RegistracijaProvajdera {

    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
       Provider prov = new org.bouncycastle.jce.provider.BouncyCastleProvider();
       Security.addProvider(new BouncyCastleProvider());
       System.out.println("Provajder:" + prov.getInfo());
    }
}
```

*Primer 1.1 Dinamička registracija provajdera*

Programski izlaz:

```
Provajder:BouncyCastle Security Provider v1.51
```

U primeru (Primer 1.2), pokazaćemo na koji način da izlistamo sve podržane jednosmerne funkcije (*MessageDigest*) u *BouncyCastle* provajderu. Ukoliko želimo da prikažemo sve podržane algoritme iz šifarske klase, umesto stringa „MessageDigest“ unećemo „Cipher“. Cipher je klasa iz Java kriptografske arhitekture. Klasa obezbeđuje metode za upravljanje sa šifarskim algoritmima.

```
public class RegistracijaProvajdera {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
       Provider prov = new org.bouncycastle.jce.provider.BouncyCastleProvider();
       Security.addProvider(new BouncyCastleProvider());
       System.out.println("Provajder:" + prov.getInfo());
       System.out.println();
       //Prikaz dostupnih jednosmaernih funkcija u BouncyCastle provajsderu
       Provider[] providers = Security.getProviders();
       Set<String> MessageDigest = new HashSet<String>();
       for (int i = 0; i != providers.length; i++) {
           Iterator it = providers[i].keySet().iterator();
           while (it.hasNext()) {
             String entry = (String) it.next();
             if (entry.startsWith("MessageDigest.")) {
                 MessageDigest.add(entry.substring("MessageDigest.".length()));
             }else{
                 System.out.println(entry);
             }
           }
       }
       printSet("Lista jednosmernih heš funkcija", MessageDigest);
    }

    public static void printSet(String setName, Set algoritam) {
        System.out.println(setName + ":");
        if (algoritam.isEmpty()) {
            System.out.println("Nije dostupno.");
        } else {
            Iterator it = algoritam.iterator();
            while (it.hasNext()) {
                String ime = (String) it.next();
                System.out.println("            " + ime);
            }
        }
    }
}
```

*Primer 1.2 Generisanje listinga za „MessageDigest“ grupu algoritama*

Programski izlaz:

```
Lista jednosmernih heš funkcija:
            SHA-1
            SHA-384
            MD5 ImplementedIn
            Skein-256-128
            Skein-512-512
            SHA
            Skein-512-256
            Tiger
            WHIRLPOOL
            Skein-512-384
            GOST3411
            Skein-256-256
            Skein-512-160
            SHA-224
            SM3
            SHA3-384
            SHA-512/224
            SHA ImplementedIn
            TIGER
            RIPEMD256
            SHA3-224
            SHA3-512
            RIPEMD320
            RIPEMD160
            Skein-512-128
            Skein-1024-512
            Skein-512-224
            Skein-1024-384
            Skein-256-224
            SHA-512/256
            SHA-256
            MD2
            RIPEMD128
            Skein-1024-1024
            MD4
            SHA3-256
            SHA-512
            Skein-256-160
            MD5
```

Neke Java klase koje ćemo kasnije koristiti u primerima nisu podržane od strane standardnih provajdera za *JCA* i *JCE*, tako da je za potrebe laboratorijskog neophodno registrovati i druge kriptografske provajdere. U primerima koji kasnije slede, navešćemo svaki put kada uvedemo novi provajder u upotrebu.

U primeru (Primer 1.3), videćemo kako jednostavno možemo da prikažemo listu svih podržanih kriptografskih servisa od unutar *BouncyCastle* provajdera.

```
public class BClistaServisa {
    public static void main(String[] args) {
        Security.addProvider(new BouncyCastleProvider());
        Set<Object> rezultat = new HashSet<>();
        Provider[] providers = Security.getProviders();
        for (Provider provider : providers) {
            Set<Object> keys = provider.keySet();
            for (Object key : keys) {
                String data = (String) key;
                data = data.split(" ")[0];
                if (data.startsWith("Alg.Alias")) {
                    data = data.substring(10);
                }
                data = data.substring(0, data.indexOf('.'));
                rezultat.add(data);
            }
        }
        for (Object o : rezultat) {
            System.out.println("Tip servisa = " + o);
        }
    }
}
```

*Primer 1.3 Generisanje liste dostupnih kriptografskih servisa*

Prikazana su dva izlaza iz program. Prvi izlaz predstavlja listu servisa dobijenih za standardni provajder, tj. registrovan prilikom prve instalacije *JVM*. Standardni provajder je *SunJCE* i u tom slučaju potrebno je prvu liniju koda označiti kao komentar (//).

Drugi izlaz predstavlja listu servisa dobijenih za naknadno dinamički registrovani provajder *BouncyCastle*.

Komparativnom analizom ove dve liste, možemo da utvrdimo da provajder *BouncyCastle* obezbeđuje podršku za još jedan dodatni servis – `X509Store`- servis za upravljanje digitalnim sertifikatima.

Izlaz iz obe verzije programa:

**SunJCE**

```
Tip servisa = Policy
Tip servisa = Configuration
Tip servisa = AlgorithmParameterGenerator
Tip servisa = TransformService
Tip servisa = CertificateFactory
Tip servisa = KeyInfoFactory
Tip servisa = CertPathBuilder
Tip servisa = MessageDigest
Tip servisa = KeyAgreement
Tip servisa = KeyGenerator
Tip servisa = SecretKeyFactory
Tip servisa = KeyFactory
Tip servisa = XMLSignatureFactory
Tip servisa = SaslServerFactory
Tip servisa = TerminalFactory
Tip servisa = SecureRandom
Tip servisa = SaslClientFactory
Tip servisa = KeyPairGenerator
Tip servisa = SSLContext
Tip servisa = KeyStore
Tip servisa = Mac
Tip servisa = Provider
Tip servisa = KeyManagerFactory
Tip servisa = CertPathValidator
Tip servisa = Signature
Tip servisa = TrustManagerFactory
Tip servisa = Cipher
Tip servisa = CertStore
Tip servisa = GssApiMechanism
Tip servisa = AlgorithmParameters
```

**BouncyCastle**

```
Tip servisa = Policy
Tip servisa = X509Store
Tip servisa = Configuration
Tip servisa = AlgorithmParameterGenerator
Tip servisa = TransformService
Tip servisa = CertificateFactory
Tip servisa = KeyInfoFactory
Tip servisa = CertPathBuilder
Tip servisa = MessageDigest
Tip servisa = KeyAgreement
Tip servisa = KeyGenerator
Tip servisa = SecretKeyFactory
Tip servisa = KeyFactory
Tip servisa = XMLSignatureFactory
Tip servisa = SaslServerFactory
Tip servisa = TerminalFactory
Tip servisa = SecureRandom
Tip servisa = SaslClientFactory
Tip servisa = KeyPairGenerator
Tip servisa = SSLContext
Tip servisa = KeyStore
Tip servisa = Mac
Tip servisa = Provider
Tip servisa = KeyManagerFactory
Tip servisa = X509StreamParser
Tip servisa = CertPathValidator
Tip servisa = Signature
Tip servisa = TrustManagerFactory
Tip servisa = Cipher
Tip servisa = CertStore
Tip servisa = GssApiMechanism
Tip servisa = AlgorithmParameters
```

# 2. KRIPTOLOŠKI KLJUČEVI I IMPLEMENTACIJA SIMETRIČNIH ŠIFRI

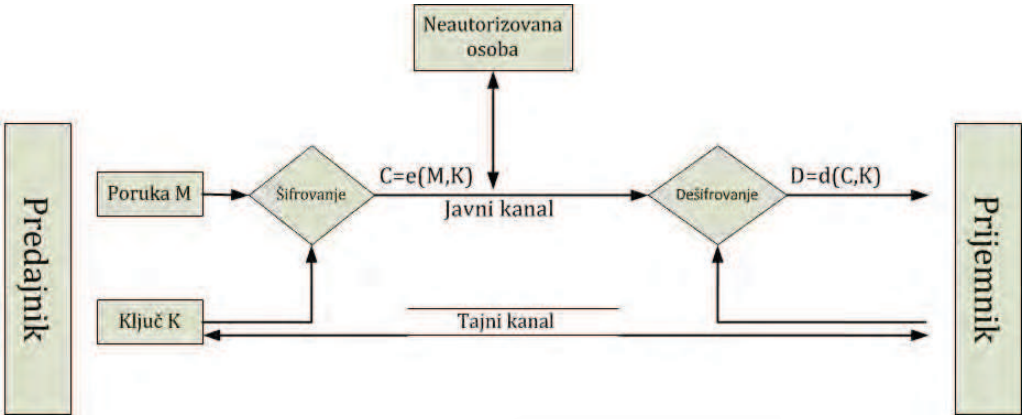
Nakon što smo razumeli kako *JCA* i *JCE* obezbeđuju neophodnu infrastrukturu za primenu kriptografskih operacija u Javi, spremni smo da istražimo sve funkcionalnosti za pravilno konstruisanje simetričnih šifri, ali pre nego što dođemo do prvog primera u kome ćemo demonstrirati korišćenje simetrične šifre, napravićemo jedan teorijski osvrt na simetrične blokovske šifre.

U klasičnoj kriptografiji koriste se takozvani „papir olovka algoritmi“. Koristio se isti ključ za šifrovanje i dešifrovanje (simetrični ključ). Otvoreni tekst i šifrat se uglavnom predstavljao sa slovima. Krajem Drugog svetskog rata, uvedeni su računari u kriptoanalizu, što je dalje indukovalo i kod kriptografa upotrebu računara za šifrovanje i dešifrovanje.

Moderni algoritmi podrazumevaju fazu kodovanja, otvoreni tekst se pre šifrovanja prvo predstavi preko binarnih nizova ili bitova. Način kodovanja nije tajna, koristi se ASCII tabela. ASCII tabela je skraćenica od „Američki standardni kod za razmenu podataka”. Algoritmi za šifrovanje i dešifrovanje sada rade na nizovima bita, a tako smo dobili digitalno šifrovanje.

Šifrovanje pomoću računara je u osnovi slično klasičnim oblicima šifrovanja. I dalje se koriste kombinacije transpozicije „premeštanja“ i supstitucije „zamene“. Postoje brojne prednosti moderne kriptografije u odnosu na klasičnu kriptografiju od kojih izdvajamo dve najbitnije. Prva prednost je u tome da je moguće realizovati daleko složenije šifre primenom računara za razliku od onih koje se realizuju mehaničkim putem. Druga, elektronika je neuporedivo brža od mehanike što utiče na brzinske performanse algoritama za šifrovanje i dešifrovanje.

Blokovske šifre se mogu posmatrati kao moderni naslednici klasičnih kodnih knjiga gde ključ određuje kodnu knjigu. Unutrašnji rad algoritama savremenih blokovskih šifara je kompleksan. Kod blokovskih šifara, otvoreni tekst se deli na blokove fiksne i zadate dužine (64, 128, 192, 256) u bitovima, zatim se svaki blok nezavisno šifruje. Interno, blokovske šifre primenjuju i konfuziju i difuziju.




*Slika 2.1 Princip rada simetrične šifre*

Blokovske šifre kombinuju zajedno svojstvo difuzije, svojstvo konfuzije i kompletnosti. Blok otvorenog teksta odredićemo sa *Pi*, a blok šifrata *Ci*. Difuzija podrazumeva da poznavanje para *Pi* i *Ci* ne sme da omogući da se na osnovu *Cj* odredi odgovarajuće *Pj*. Kod blokovskih šifara zahtevano je da male promene u bloku otvorenog teksta (1 bit), izazovu nepredvidive promene u datom bloku šifrata. Svojstvo konfuzije podrazumeva da će svi ključevi biti jednako verovatni u slučaju napada potpunom pretragom ključeva. Na kraju, celovitost podrazumeva da će svaki bit šifrata postati funkcija svakog bita ključa.

Ciljevi dizajna blokovskih šifri su sigurnost i efikasnost. Nije toliko teško razviti razumno sigurnu blokovsku šifru ili efikasnu blokovsku šifru, ali dizajnirati nešto što je i sigurno i efikasno, zahteva visok stepen kriptografskog umeća. Blokovske šifre se najčešće realizuju softverski.

Simetrični šifre i skoro svi drugi šifarski mehanizmi u kriptografiji zahtevaju pouzdan generator brojeva. Određena količina slučajnih brojeva predstavlja materijal za izgradnju kriptološkog ključa. Pod slučajnim brojevima podrazumevamo samo čisto slučajne brojeve koje je moguće dobiti samo preko nekog prirodnog izvora informacije. U potreba pseudo slučajnih brojeva u ove svrhe nije dozvoljena, jer na taj način protivnik bi mogao uz vrlo mali broj pokušaja da reprodukuje istu sekvencu brojeva i tako dođe do otvorenog teksta. Pre nego što konstruišemo prvu blokovsku

šifru preko *JCE* proširenja u Javi, prvo ćemo se osvrnuti na slučajne brojeve koje je moguće generisati preko *JCA* proširenja u Javi.

## 2.1 GENERISANJE SLUČAJNIH BROJEVA

Kao što smo već rekli u prethodnom poglavlju o značaju slučajnih brojeva i načinima da se dođe do njih, mi nemamo mogućnost da tako jednostavno povežemo naš računar na kome obavljamo kriptografske operacije na neki prirodni izvor informacije koji bi iskoristili za generisanje simetričnog ključa. Iz tog razloga i dan danas sva istraživanja na ovom polju su usmerena na upotrebu računara i softvera za generisanje „kvalitetnih" pseudo slučajnih brojeva.

Danas, skoro na svim softverskim platformama lako je pronaći algoritam koji treba da obezbedi generisanje kriptografskih pseudo slučajnih brojeva ili skraćenica *CSPRNG* (engl. *Cryptographically Secure Pseudo-Random Number Generator*). Međutim, ne smemo da zaboravimo Kerkehove principe, koji kažu da se čitava tajnost jednog sistema oslanja na kvalitet kriptološkog ključa, a ne na sam algoritam, ali uglavnom se pretpostavlja da kriptoanalitičar pokušava da dešifruje šifrat na osnovu svih dobijenih detalja o korišćenoj šifri.

Polazimo od pretpostavke da je šifrat jak samo u slučaju ako je za šifrovanje korišćen tajni ključ za šifrovanje poruke. Veličina prostora ključeva predstavlja broj mogućih kombinacija koje je moguće koristiti u izabranom šifarskom algoritmu. Na primer, ako je veličina ključa 64 bita, tada je prostor ključeva ili broj kombinacija $2^{64}$. Treba imati na umu da povećanjem ključa za jedan bit, ostvarujemo eksponencijalni rast prostora ključeva i da tako smanjujemo mogućnost za razbijanje neke šifre.

Na primer, 2006. je napravljena nova mašina *COPACABANA* od strane tima sa Univerziteta Bohumu i Kil u Nemačkoj. Za razliku od drugih mašina ova mašina je komercijalno dostupna. U 2008. godini razvijena je nova verzija, *COPACABANA RIVYERA* koja je smanjila vreme potrebno za razbijanje DES-a na manje od jednog dana. Ona koristi 128 Spartan-3 5000 *FPGA* čipove. Trenutno *SciEngines* kompanija u Nemačkoj *RIVYERA* drži rekord u razbijanju *DES*-a metodom potpune pretrage ključeva. Jedan od zanimljivijih aspekata projekta *COPACABANA* je količina potrebnog

novca za izgradnju takvog hardvera. Jedna mašina može se napraviti za 10.000 dolara.

U nastavku razmotrićemo i primenu lozinki u ulozi tajnog kriptološkog ključa. Šta je zapravo stvarana dužina lozinke? Ukoliko koristimo velika slova, mala slova i brojeve, tada ćemo ostaviti na korišćenje samo 62 znaka ([A… Z], [a… z], [0… 9]).

Ukoliko pretpostavimo da je naša lozinka **StuDent**, u tom slučaju ako je kriptoanalitičaru poznat skup korišćenih karaktera, broj kombinacija je 62*62*62*62*62*62*62, gde svaki karakter može da bude na bilo kojoj poziciji, u tom slučaju imamo 3,521,614,606,208 kombinacija. Snagom današnjih procesora, ovaj broj ne predstavlja veliki izazov. Međutim, ako se pretpostavi da je lozinka sadržana u rečniku, tada je razbijanje lozinke od 7 karaktera još manje vremenski zahtevno.

Upravo smo nagovestili problem koji je prisutan kod lozinki, ljudi nemaju mogućnost da zapamte kompleksne lozinke koje bi bile dovoljnog kvaliteta za šifrovanje podataka. Iz tog razloga *JCA* proširenje uključuje Java klasu `SecureRandom` koju je moguće upotrebiti upravo u tu svrhu.

## 2.2 GENERISANJE KRIPTOGRAFSKIH PSEUDO SLUČAJNIH BROJEVA

Generisanje kriptografskih pseudo slučajnih brojeva predstavlja mogućnost od velikog značaja (*CSPRNG*). Za ove potrebe *JCA* proširenje je stavilo na dohvat ruke funkcionalnosti za upotrebu algoritama za *CSPRNG*.

Sledi primer sa instanciranjem objekta *CSPRNG* preko klase `SecureRandom`:

```
SecureRandom csprng = SecureRandom.getInstance("SHA1PRNG");
```

Nakon instanciranja objekta, neophodno je zadati inicijalno stanje unutrašnjem algoritmu. Pseudo slučajni generatori su algoritmi koji na osnovu početnih (inicijalnih) stanja daju izlazni niz pseudo slučajnih brojeva. Početni uslov je zadovoljen ukoliko je ulaz u generator „istinski“ slučajan.

```
//slučajno otkucani brojevi

csprng.setSeed(31592712351);
```

Početni „istinski“ slučajan niz se dobija primenom generatora „istinski“ slučajnih brojeva TRNG i predstavlja ključ *k* odnosno inicijalno stanje. Generisane pseudo slučajne sekvence su znatno duže od ključa i smatraju se pouzdanim ukoliko zadovoljavaju određene statističke testove. Zanimljivo je da se pseudo slučajni brojevi ponekad pokazuju uspešnijim od slučajnih brojeva dobijenih iz fizičkih izvora. Pseudo slučajni generatori *PRNG* pokazuju dobre karakteristike uz veoma oprezan odabir početnih stanja što dovodi do prikrivanja determinističkih osobina koje se mogu upotrebiti za kriptoanalizu.

Izdvajanjem slučajnih bajtova preko objekta *SCPRNG* se vrši pozivanjem funkcije `nextBytes()`, čiji je rezultat predstavljen sa nizom bajtova. Nakon jednog poziva funkcije, važno je promeniti inicijalno stanje algoritma preko funkcije `setSeed()`, da bi se izbeglo ponavljanje identičnog niza bajtova i tako sprečili algoritam da ode u stanje determinizma koje nikako ne bi bilo korisno u kriptografskim svrhama. U protivnom kriptoanalitičar bi mogao vrlo lako da pogodi redosled bajtova.

Takođe, važno je napomenuti da `java.lang.Math` klasa sadrži sličnu funkciju, funkciju `random()`, koja nije pogodna za kriptografske operacije, jer koristi deterministički algoritam i time proizvodi determinističke nizove bajtova. U Java dokumentaciji, navedeno je da su prave slučajne vrednosti uniformno distribuirane između 0,0 i 1,0. Uniformnost je vrlo važna osobina koju poseduju generisane sekvence, jer na taj način jačamo svoju poziciju pred kriptoanalitičarem. O značaju uniformnosti pogledati udžbenik „Kriptologija 1“, u kome je detaljnije obrazložena teorija kvalitetnih slučajni sekvenci sa aspekta teorije informacija i primene informacionih mera u analizi kvaliteta slučajnih brojeva.

### 2.2.1 GENERISANJE SLUČAJNE VREDNOSTI

Jedna od najlepših karakteristika *JCA* je `SecureRandom` klasa, koja zahteva dodatne metode preko kojih će obezbediti kvalitetne slučajne bajtove podataka. `SecureRandom` klasa predstavlja proširenje `java.util.random` klase. Iako,

sama po sebi klasa `java.util.Random` nije pogodna za kriptografske operacije. Međutim, ukoliko je korišćen argument *SHA1PRNG*, koji upućuje na upotrebu nedeterminističkog algoritma za generisanje slučajnosti, u tom slučaju `SecureRandom` klasa postaje prihvatljiva za upotrebu u kriptografskim algoritmima.

*Tabela 2.1 Generisanje pseudo slučajne vrednosti*

| Naziv algoritma | Opis |
|---|---|
| **SHA1PRNG** | Naziv pseudo slučajni generator brojeva *PRNG* dobijen je od *SUN* provajdera. Ovaj algoritam koristi *SHA*-1 kao osnovu za *PRNG*. Izračunava *SHA*-1 heš vrednost nad čisto slučajnim inicijalnim stanjem (engl. *seed*). Od 160 izlaznih bitova SHA-1 vrednosti, koriste se samo 64 bita. |

Razumevanje načina za generisanje slučajnih brojeva je od izuzetnog značaja bezbednu primenu kriptografskih algoritama. Nakon toga možemo da pređemo na pravila i principe koji su jasno utvrđeni, a tiču se bezbedne implementacije algoritama za generisanje kriptološkog ključa za primenu u simetričnim šiframa.

```
public class GeneratorSlucajnihBrojeva {
    public static void main(String[] args) throws NoSuchAlgorithmException {
        //Deklarisanje i instanciranje objekta scprng
        SecureRandom csprng = SecureRandom.getInstance("SHA1PRNG");

        //Inicijalno stanje unutrašnjeg algoritma,
        //Koristimo trenutno vreme u mili sekundama
        csprng.setSeed(System.currentTimeMillis());

        //Generisanje slučajnog logičkog tipa vrednosti
        boolean slucajanLogickiTip = csprng.nextBoolean();

        //Ispis generisane vrednosti
        System.out.println("Boolean: "+slucajanLogickiTip);

        //Generisanje slučajnog celobrojnog tipa vrednosti
        int slucajanCelobrojniTip = csprng.nextInt();

        //Ispis generisane vrednosti
        System.out.println("Integer: "+slucajanCelobrojniTip);

        //Generisanje 5 clučajnih bajt vrednosti
        byte bajtovi[] = new byte[5];
        csprng.nextBytes(bajtovi);
        //Ispis generisane vrednosti
        for (byte i : bajtovi) {
            System.out.println("Bajt: "+i);
        }
    }
}
```

*Primer 2.1 Generator slučajnih brojeva*

Programski izlaz:

```
Boolean: true
Integer: 2056140257
Bajt: -101
Bajt: -85
Bajt: -26
Bajt: -113
Bajt: -56
```

U Java primeru (Primer 2.1) prvo inicijalizujemo objekat *CSPRNG,* zahtevajući algoritam (*SHA1PRNG*). U drugom koraku generišemo inicijalnu vrednost preko funkcije za računanje trenutnog vremena u mili sekundama. Dobijena vrednost je tipa

*LONG* čija je veličina 64 bita. Prvo smo generisali logički tip vrednosti, zatim celobrojni tip vrednosti i niz bajtova kao slučajno generisane vrednosti. Sve generisane vrednosti su na kraju ispisane. Ako bismo želeli da i dalje koristimo inicijalno stanje za generisanje slučajnih vrednosti, bilo bi poželjno da promenimo inicijalno stanje pozivajući ponovo funkciju `setSeed()`.

## 2.3 GENERISANJE KRIPTOLOŠKOG KLJUČA

Sve simetrične šifre zahtevaju tajni kriptološki ključ, koji će koristiti za operaciju šifrovanja ili dešifrovanja. Kao što smo prethodno rekli, tajni ključ predstavlja kritičnu tačku svakog simetričnog kripto sistema. Uvek pretpostavljamo da kriptoanalitičar ima sve detalje koji se odnose na specifikaciju korišćenog algoritma za šifrovanje. *JCA* arhitektura delegira odgovornost za tajne ključeve simetričnih šifara na `javax.crypto.KeyGenerator` klasu. Ova klasa implementira par metoda za generisanje tajnog ključa.

U tabeli koja sledi, nalaze se algoritmi koji mogu biti zahtevani pri kreiranju instance `KeyGenerator`.

*Tabela 2.2 Algoritmi za generisanje simetričnog ključa*

| Naziv algoritma | Opis |
|---|---|
| **AES** | Generator ključa za *AES* algoritam.. |
| **ARCFOUR** | Generator ključa za *ARCFOUR* (*RC4*) algoritam. |
| **Blowfish** | Generator ključa za *Blowfish* algoritam. |

| | |
|---|---|
| **DES** | Generator ključa za *DES* algoritam. |
| **DESede** | Generator ključa za *DESede* (*3DES*) algoritam. |
| **HmacMD5** | Generator ključa za *HmacMD5* algoritam. |
| **HmacSHA1 HmacSHA256 HmacSHA384 HmacSHA512** | Generator ključa za različite tipove *HmacSHA* algoritama. |
| **RC2** | Generator ključa za *RC2* algoritam. |

Sledeća linija koda predstavlja kreiranje instance ključa za AES algoritam:

```
KeyGenerator kg = KeyGenerator.getInstance("AES");
```

Kriptografija kao nauka o dizajniranju šifri bila bi isuviše ograničena ukoliko bi šifarski algoritam bio inicijalizovan samo sa jednim parametrom. Međutim, različite šifre koriste različite inicijalizacione parametre, da bi implementirali sve funkcionalnosti izabrane šifre.

Na primer, neke simetrične šifre mogu da zahtevaju veličinu ključa 128 bitova, dok druge mogu da zahtevaju 192 ili 256 bitova. Upravo, klasa `KeyGenerator` enkapsulira ovaj koncept za nas, tako što obezbeđuje informaciono kvalitetne kriptološke ključeve koji će biti u stanju da zadovolje zahteve bilo koje simetrične šifre.

`KeyGenerator` klasa prati originalni dizajn *JCA* arhitekture, pružajući potpuno nezavisan algoritam za generisanje tajnih ključeva. Ovakva funkcionalnost je postignuta preko niza preopterećenih metoda `init()`. Na samom početku nismo u

obavezi da izaberemo nijedan od ponuđenih, već ta procedura može biti odložena do samog kraja. U nastavku razmotrićemo svaki scenario.

### 2.3.1 NEZAVISNO INICIJALIZOVANI ALGORITMI

Odluka da se koristi nezavisno inicijalizovani algoritmi je slučaj koji omogućava aplikaciji da pretražuje šifarske algoritme koji će se koristiti u trenutku izvršavanja. Bilo koja od ovih metoda prihvata argumente tipa String. Obzirom da svaki provajder definiše sopstvenu listu dostupnih algoritama, korisniku se pruža mogućnost da napravi vlastiti izbor šifarskih algoritama u trenutku izvršavanja. Ova situacija bi zahtevala neki algoritam nezavisan od procesa inicijalizovanja tajnog ključa. Sledeći preopterećeni `init()` metodi podržavaju ovaj pristup u radu:

```
init(int veličinaKljuča)

init(SecureRandom izvorSlučajnosti)

init(int veličinaKljuča, SecureRandom izvorSlučajnosti)
```

Prvi `init`() metod određuje veličinu ključa (veličina će biti određena u bitovima ili bajtovima u zavisnosti od algoritma koji će se koristiti). Provajder snosi odgovornost za lociranje izvora slučajnosti. Većina provajdera se jednostavno oslanja na provajder sa najvećim prioritetom koji implementira algoritam SHA1PRNG preko `SecureRandom` klase.

Ukoliko programer želi da koristi drugi algoritam za CSPRNG, drugi metod `init`() to omogućava jednostavno uzimanjem specifičnog izvora slučajnosti – prosleđivanjem argumenta u vidu `SecureRandom` instance. U ovom slučaju dužina ključa će biti podrazumevana. Na ovaj način smo samostalno odredili šta će se koristiti kao izvor slučajnosti.

Poslednji `init`() metod je potpuno nezavisan algoritam, za razliku od prethodna dva. U njemu definišemo oba argumenta (veličinu ključa i izvor slučajnosti).

### 2.3.2 SPECIFIČNO INICIJALIZOVANI ALGORITMI

Ovaj pristup se koristi u situacijama kada želite definisati posebno algoritam koji će se koristiti i sve njegove specifične parametre ključne za inicijalizovanje. Ovaj pristup olakšava `KeyGenerator` klasa, tako što obezbeđuje dva dodatna preopterećena `init()` metoda koja pružaju neophodnu funkcionalnost u vidu podrške:

```
init(AlgorithmParameterSpec parametri)

init(AlgorithmParameterSpec parametri, SecureRandom
izvorSlučajnosti)
```

Svaki od ovih `init()` metoda zahtevaju upotrebu java klase `AlgorithmParameterSpec`, koja se nalazi u paketu javax.`crypto`.spec. Na osnovu uvida u Java dokumentaciju, uvideli smo da je klasa `AlgorithmParameterSpec` samo klasa interfejs. Ovaj interfejs definiše formalne javne metode ili statičke konstante, identične konceptu koji se uglavnom i koristi u `java.io.Serializable` interfejsu. Ova arhitektura praktično dozvoljava provajderu da definiše bilo koju klasu kao zaseban algoritam sa specifičnim parametrima, jednostavno uključujući `AlgorithmParameterSpec` `klasni` interfejs.

Prva `init()` metoda je slična sa `init(int veličinaKljuča)`, gde je provajder odgovoran za korišćeni izvor slučajnosti, dok drugi `init()` metod diktira poseban izvor slučajnosti.

### 2.3.3 GENERISANJE SIMETRIČNOG KLJUČA PREKO SECRETKEY KLASE

Svaka šifra može imati različite zahteve kada je upitanju generisanje tajnog ključa. Iz tog razloga, ključevi su skriveni iz niza interfejsa. Najvišeg nivoa interfejs je ključ interfejs, a to definiše tri sledeće metode:

*Tabela 2.3 Interfejsi za rad sa ključevima*

| Naziv metoda | Opis |
|---|---|
| **getAlgorithm()** | Vraća String naziv šifarskog algoritma. |
| **getFormat()** | Vraća String naziv formata kodovanog materijala za ključ. |
| **getEncoded()** | Vraća niz byte[] koji predstavlja ključ u kodovanom format. |

Klasa `KeyGenerator` uvek vraća kao rezultat `SecretKey` instancu, na primer:

```
public class GenerisanjeKljuca {
    public static void main(String[] args) throws NoSuchAlgorithmException {
        //Inicijalizovanje instance KeyGenerator za AES algoritam
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        //Inicijalizovanje instance SecretKey
        SecretKey key = kg.generateKey();
        //Dostupni interfejsi za instancu ključa
        //Ispis šifarskog algoritma
        System.out.println("Šifarski algoritam: "+key.getAlgorithm());
        //Ispis korišćenog formata za kodovanje
        System.out.println("Format kodovanja: "+key.getFormat());
        //Ispis generisanih bajtova ključa
        for(byte i:key.getEncoded()){
            System.out.print(i+"  ");
        }
        System.out.println("");
        //Ispis veličine ključa u bitovima
        System.out.println("Veličina ključa je "+(key.getEncoded().length*8));
    }
}
```

*Primer 2.2 Generisanje tajnog ključa*

Programski izlaz:

```
Šifarski algoritam: AES
Format kodovanja: RAW
-116  14  77  26  -94  33  99  9  -74  -8  -38  97  -118  56  -110  -13
Veličina ključa je 128
```

Kao što možemo da vidimo u primeru (Primer 2.2), `SecretKey` instanca nema više javnih metoda ili konstanti izvan ovih koje smo koristili, a koristi se isključivo za simetrične tajne ključeve.

Pored navedenih metoda, bitno je zapaziti da postoji metod `destroy()` koji se koristi za uništavanje `SecretKey` instance na potpuno bezbedan način.

## 2.4 TRANSPARENTNOST GENERISANIH KLJUČEVA

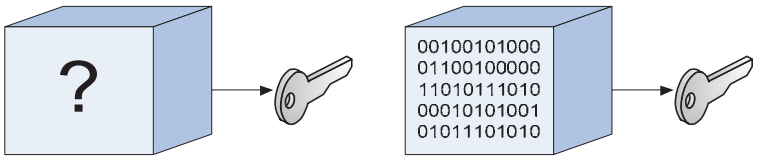
Rezultat metoda `generateKey()` nad klasom `KeyGenerator` je provajderski specifičan (Primer 2.2) i predstavlja način za generisanje netransparentog ključa. Kada kažemo da ključ poseduje osobinu netransparentnosti, mislimo na ključ o kome mi ne znamo mnogo detalja i tada možemo da ga poredimo sa osobinama koje poseduje crna kutija (poznat ulaz, poznat izlaz, ali ne i unutrašnjost kutije). Jedino što mi znamo, jeste da je ključ generisan. Ovo je postignuto zahvaljujući posebnom dizajnu klase `SecretKey`, jer u njoj ne postoji mehanizam za pronalaženje meta podataka o generisanom ključu. U slučaju da nas interesuje kvalitet ključa, ne bismo umeli da odgovorimo na pitanje da li je ključ jak ili slab.

Da bi se obezbedila mogućnost da saznamo nešto više detalja o ključu, pored tri viđene vrste podataka koje obezbeđuje `SecretKey` klasa, *JCE* je ponudio dva dodatna mehanizma za konvertovanje `SecretKey` instance po specifikaciji za simetrični tajni ključ, a koja je nezavisna od provajdera.

1. Prvi način je kombinacijom `SecretKeyFactory` klase i još jedne klase koja implementira klasu `KeySpec` interfejs.
2. Drugi način je preko javax.crypto.cpec.`SecretKeySpec` klase.

Konvertovanjem tajnog ključa u „specifikaciju ključa“, nivo transparentnosti je postignut. Transparentnost ključa nam obezbeđuje mogućnost da sagledamo sve

meta podatke ključa. U primeru koji sledi videćemo implementaciju ključa kod DES šifarskog algoritma, jer ovakva vrsta transparentnosti nam može garantovati da je izabran stvarno entropijski kvalitetan ključ.




*Slika 2.2 Levo –netransparentni ključ, desno – transparentni ključ*

### 2.4.1 PRETVARANJE KLJUČA PO SPECIFIKACIJI – NAČIN 1

Provajder ima mogućnost direktno da koristi klasni interfejs `KeySpec`, ali najčešće je korišćena specifična klasa za implementaciju klasnog interfejsa `KeySpec`. Mnogi primeri se nalaze u `java.security.spec` paketu. Na primer, ovaj paket definiše `DESKeySpec` klasu, koja opisuje DES ključ nezavisno od provajdera.

Ona u sebi sadrži nekoliko metoda koji se koriste za pristup meta podacima ključa. Na primer, tu se nalazi metod koji može da obezbedi informaciju o kvalitetu ključa, da li je ključ jak, slab ili ga možda treba odbaciti.

Zapravo, izlaganje meta podataka ključa čini specifikaciju ključa. Na ovaj način obezbeđena je mogućnost priključivanja sa različitim provajderima za različite kriptografske operacije. Na primer, *DES* ključ generisan preko klase `SecureRandom`, preko *CSPRNG* algoritma, mogao bi da se koristi u *DES* simetričnoj šifri drugog provajdera. U nekim situacijama ovo se dešava u pozadini i bez eksplicitnog znanja.

Po definiciji, kada provajder nije eksplicitno naveden, tada se regularne dostupne metode oslanjaju na deklaracije provajdera u `java.security` paketu. Stoga, nije teško zamisliti situaciju u kojoj `KeyGenerator` dolazi iz jednog provajdera, a `Cipher` - šifarska klasa iz nekog drugog.

Zadatak svakog provajdera je da dokumentuje podržane specifikacije ključa. U primeru koji sledi, generisaćemo ključeve za *DES* simetričnu šifru koristeći provajder sa najvišim prioritetom koji implementira *DES* generator ključa.

```
public class KonvertovanjeKljuca1 {
    public static void main(String[] args) throws InvalidKeyException, NoSuchAlg
        //Slučajno izabrani bajtovi, ili može biti korišćena SecureRandom klasa
        //koja dolazi preko provajdera u koji nemamo poverenje
        byte kljuc1[] = {114, -125, 96, -76, 23, -113, -95, -106, 55, -11, 123};
        DESKeySpec specifikacijaDESkljuca = new DESKeySpec(kljuc1);

        //Preko DES specifikacije ispisujemo potrebnu dužinu ključa
        System.out.println("DES veličina ključa: "+DESKeySpec.DES_KEY_LEN);

        //Zatim, proveravamo kvalitet ključa, prvi parametar je ključ, a drugi
        //početna pozicija iz niza (indeksna pozicija u nizu bajtova)
        if(DESKeySpec.isWeak(kljuc1, 0))
            System.out.println("Ključ je slab!");
        else
            System.out.println("Ključ je jak!");

        //Kreiranje instance za fabriku ključeva
        SecretKeyFactory fabrika = SecretKeyFactory.getInstance("DES");

        //Ključ je spreman i može da se stavi u funkciju DES šifre
        SecretKey kljuc2 = fabrika.generateSecret(specifikacijaDESkljuca);

        //Prikaz ključa
        for(byte i:kljuc2.getEncoded())
               System.out.print(i+ " ");
        System.out.println("");
    }
}
```

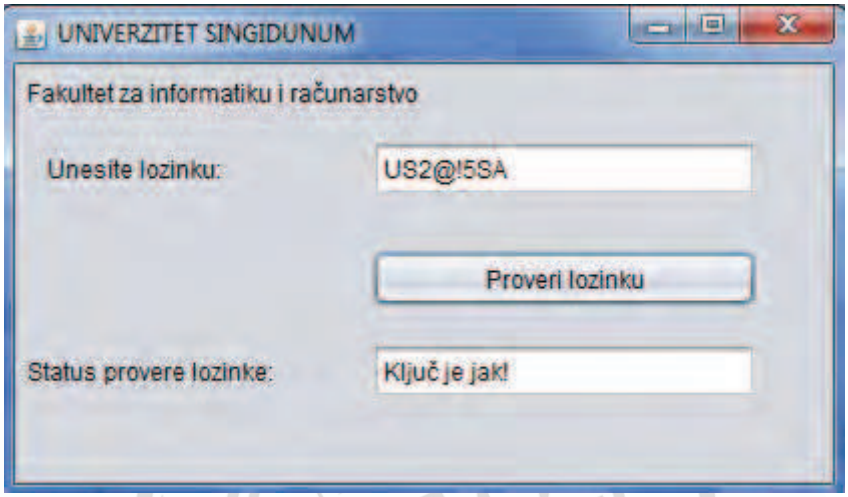
*Primer 2.3 Konvertovanje ključa – način 1*

Programski izlaz:

```
DES veličina ključa: 8
Ključ je jak!
115 -125 97 -75 22 -113 -95 -105
```

U nastavku sledi primer u kome ćemo testirati kvalitet lozinki koje čovek pretpostavimo može da smisli na osnovu slučajnosti ljudske prirode. Tako dobijene String-ove konvertovaćemo u niz bajtova preko funkcije `getbyte()`. U ovom

slučaju ne koristimo klasu `KeyGenerator` klasu. Napravićemo *GUI* interfejs da bi omogućili korisniku da preko grafičkog interfejsa unese smišljenu lozinku. Osnovna ideja je da aplikacija proveri kvalitet lozinke koja će se koristiti kao ključ u *DES* algoritmu.




*Slika 2.3 Provera kvaliteta simetričnog ključa*

Da podsetimo, upotreba klase koja implementira `KeySpec` klasni interfejs, omogućava provajderu i nama transparentan pristup ključu. Ova transparentnost daje mogućnost za dalje ispitivanje ključa i njegovih meta podataka, da bi se utvrdilo da li ispunjava određeni kvalitet za upotrebu u određenom algoritmu.

```
private void provera_bActionPerformed(java.awt.event.ActionEvent evt) {

    try {
        byte kljuc1[] = this.lozinka_tf.getText().getBytes();
        DESKeySpec specifikacijaDESkljuca = new DESKeySpec(kljuc1);
        if(DESKeySpec.isWeak(specifikacijaDESkljuca.getKey(),0))
            this.ststus_tf.setText("...");
        else
            this.ststus_tf.setText("...");

    } catch (InvalidKeyException ex) {
        Logger.getLogger(ProveraKljuca.class.getName()).log(Level.SEVERE, n
    }

}
```

*Slika 2.4 Funkcionalni fragment koda* GUI *aplikacije*

Ovakva vrsta provere je vrlo bitna, obzirom da ima zanemarljiv uticaj na performanse prilikom provere ključa. Ovoj vrsti provere dajemo veliki značaj zbog postojanja dvostrukih *DES* ključeva ili duala. Postoje određeni ključevi gde za svaki ključ *K* postoji ključ $K'$za bilo koje šifrovanje sa *K* koje je identično za dešifrovanje sa $K'$ i obrnuto. Ključevi sa *K* sa $K'$su pronađeni. Dualni ključevi proizvode sve nule, sve jedinice ili nule i jedinice po nekom obrascu. Upravo takvi ključevi su navedeni u nastavku, preuzeto iz standarda *FIPS*-74.

*Tabela 2.4 Lista dvostrukih ključeva definisana u* FIPS-*74*

| **Ključ** | **Duali** |
|---|---|
| E001E00lFl0lFl0l | 01E001E00lFl0lFl |
| FElFFElFFEOEFEOE | 1FFElFFEOEFEOEFE |
| E01FE01FF10EF10E | 1FE01FEOOEF10EF1 |

| | |
|---|---|
| 01FE01FE01FE01FE | FE01FE01FE01FE01 |
| 011F011F0l0E010E | 1F011F0l0E0l0E01 |
| E0FEE0FEFlFEFlFE | FEE0FEE0FEFlFEF1 |
| 0101010101010101 | 0101010101010101 |
| FEFEFEFEFEFEFEFE | FEFEFEFEFEFEFEFE |
| E0E0E0E0FlFlFlFl | E0E0E0E0FlFlFlFl |
| lFlFlFlFOEOEOEOE | lFlFlFlFOEOEOEOE |

Kod prvih šest ključeva dualni ključevi su različiti od njih što se vidi u tabeli. Poslednja četiri ključa su identična sa svojim dualima i nazivaju se dvostruki ili dualni ključevi. Upravo su to ključevi kod kojih je dvostruko šifrovanje jednako ne šifrovanju ili otvorenom tekstu. Nijedni drugi ključevi nemaju svoje duale.

Podaci se mogu najpre dešifrovati, a zatim šifrovati (umesto šifrovati pa dešifrovati) da bi rezultirali sa otvorenim tekstom. Otvoreni tekst se može šifrovati nekoliko puta, a zatim dešifrovati isti broj puta da bismo došli do otvorenog teksta. Slično prethodnom, podaci se mogu šifrovati sukcesivno sa različitim ključevima i dešifrovati sukcesivno od strane istih ključeva da bismo dobili odgovarajući otvoreni tekst, pod uslovom da su operacije izvedene sa pravilnim redosledom. Ako je D1(E1(P)) = P (ako šifrujemo otvoreni tekst sa K1 i dešifrujemo sa K1), tada su sledeće tvrdnje istinite:

1. E1(D1(P))= P
2. E1(E1(P))= P za sebi dualne ključeve

3. D1(D1(E1(E1(P))))= P
4. E1(E1(D1(D1(P))))= P
5. D1(D2(E2(E1(P))))= P
6. D1(D2(...(D1(Ej...(E2(E1(P)...) = P
7. E1(E2(...(E1(D1...(D2(D1(P)...) = P
8. E2(E1(P)) = P za dualne ključeve
9. D2(D1(P)) = P za dualne ključeve

ali sledeći iskaz nije tačan:

10. D2(D1(E2(E,(P))))=P.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

### 2.4.2 PRETVARANJE KLJUČA PO SPECIFIKACIJI – NAČIN 2

Klasa `SecretKeySpec` može se koristiti kao prečica nezavisna od provajdera do specifikacije ključa, izbegavajući provajdera oslonjenog na klasu `SecretKeyFactory`. Realno, ova opcija je moguća za simetrične šifre koje zahtevaju jedan niz bajtova za generisanje materijala za tajni ključ. Postoje šifre koje zahtevaju više od niza bajtova, tada ćemo biti primorani da koristimo klase i metode opisane u prvom načinu. U primeru (Primer 2.5), kod koji sledi je značajno manji u odnosu na kod iz prethodnog primera.

```
package knjigazis;

import javax.crypto.spec.SecretKeySpec;

public class KonvertovanjeKljuca2 {
    public static void main(String[] args) {
        //Slučajno izabrani bajtovi, ili može biti korišćena SecureRandom klasa
        //koja dolazi preko provajdera u koji nemamo poverenje
        byte materijalKljuc[] = {96, -76, 23, -113, -95, -106, 55, -11};

        //Kreiramo tajni ključ nezavisno od provajdera za DES algoritam
        SecretKeySpec kljuc = new SecretKeySpec(materijalKljuc, "DES");

        //Ispis šifarskoh algoritma
        System.out.println("Algoritam: "+kljuc.getAlgorithm());
    }
}
```

*Slika 2.5 Konvertovanje ključa način 2.*

Uvidom u klasu `SecretKeySpec`, utvrđeno je da implementira `Key`, `KeySpec`, `SecretKey` i `Serializable` interfejs. To znači da rezultat ove operacije može da se prenosi na bilo koji metod jednog od ova tri interfejsa i da rezultat može da se serijalizuje za kasniju upotrebu. Osim toga, unos podataka je relativno jednostavan, tako da ne postoje programski izuzeci iz konstruktora koji se moraju obraditi.

Međutim, zbog jednostavnosti rukovanja, često ćemo se odreći mogućnosti da imamo uvid u ključ i meta podatke. Bitno je napomenuti da ovaj konstruktor ne proverava da li su navedeni bajtovi dovoljni da budu materijal za generisanje tajnog ključa. Na primer, ako koristimo *DES* algoritam, konstruktor ne proverava da li je ključ

8 bajtova i ne proverava kvalitet ključ (slab, polu slab). Da bi sve ove provere bile obavljene, preporučujemo uvek upotrebu klase, u ovom slučaju da bude korišćena klasa DESKeySpec.

## 2.5 KATEGORIZACIJA SIMETRIČNIH ŠIFRI

U ovom poglavlju pravimo osnovnu podelu simetričnih šifri na sekvencijalne i blokovske šifarske sisteme.

Ponavljajuća blokovska šifra deli otvoreni tekst na blokove fiksne veličine i generiše blokove šifrata fiksne veličine. U mnogim slučajevima, šifrat se dobija iz otvorenog teksta ponavljanjem funkcije *F* za određeni broj rundi. Funkcija *F*, koja zavisi od izlaza prethodne runde i ključa *K*, je poznata kao funkcija runde ne zbog svog oblika, nego zbog toga što je primenjena preko više rundi. Ulazni parametri za svaku rundu su ključ i izlaz iz prethodne runde.

Uopšteno govoreći, blok šifre su najefikasnije kada se koriste za šifrovanje jedne poruke, čija je veličina unapred poznata. Zajednička osobina svih blokovskih šifara je da su blokovi šifrata za identične blokove otvorenog teksta jednaki uz upotrebu istog ključa. Ovu ne tako dobru osobinu je moguće jednostavno prevazići uvođenjem posebnih šifarski režima. Međutim, kada se radi o protoku podataka, bez unapred poznate količine podataka, kao što je to slučaj kod *TCP/IP*, tada su sekvencijalne šifre efektivnije i efikasnije.

Sekvencijalne šifre su slične *One-time pad*-u. Zasnivaju se na generatorima slučajnih ili pseudo slučajnih nizova (radni ključevi). Generator može da koristi neki prirodni informacioni izvor i da tako obezbedi dovoljne količine radnog ključa za šifrovanje. Ovakvi generatori su čisto slučajni generatori brojeva i nemaju probleme sa periodom, ili u drugom slučaju da se koriste pseudo slučajni generatori koji koriste inicijalno stanje (engl. *Seed*) dobijeno preko prve grupe generatora, a na osnovu čega će proizvesti dovoljnu količinu radnog ključa. Ova poslednja vrsta generatora ima problem sa periodom i zbog toga je potrebno voditi računa o inicijalnom stanju i potrošenosti radnog ključa.

Osobinu periodičnosti nekog pseudo generatora možemo najlakše razumeti ako zamislimo da se krećemo kružnom putanjom i da se u jednom momentu nađemo na početku. Isto tako, pseudo generator će doći u mesto sa kog je krenuo i počeće da generiše identične vrednosti sa redosledom prema kom je prvobitno generisao. Broj pseudo vrednosti će zavisiti upravo od dužine te kružnice.

Neki autori sekvencijalne šifre nazivaju i pseudo *One-time pad* šifre. Da bi smo bolje razumeli počećemo od Šenona (*Claude Shannon*) koji je definisao dva tipa bezbednosti. Prva i najvažnija je bezuslovna bezbednost, što znači da se za zadati kriptografski sistem može rigorozno dokazati nominovana bezbednost u odnosu na protivnika koji raspolaže neograničenim vremenskim i računarskim resursima. Šifra koja odgovara datom opisu bezuslovne bezbednosti je One-time pad koja koristi radni ključ generisan preko nekog prirodnog izvora informacije i predstavlja Šenonov ideal apsolutne bezbednosti.

Druga bezbednost koju je definisao Šenon je dokaziva računarska bezbednost, a to znači da se za zadati kriptografski sistem može rigorozno dokazati nominovana bezbednost u odnosu na protivnika koji poseduje specifične vremenske i računarske resurse. Upravo šifra koja najviše odgovara definiciji računarske bezbednosti je simetrična sekvencijalna šifra i po Šenonu predstavlja pseudo ideal računarske bezbednosti (*A5/1* i *RC4*).

Bez obzira na kategoriju, simetrična šifra ima tri stvari koje treba imati na umu prilikom sprovođenja ili implementacije u softver. Tri stvari koje treba razmotriti uključuju: upravljanje ključevima, integritet podataka i neporecivost. U nastavku govorićemo o upravljanju s ključevima, dok servis neporecivosti i servis integriteta podataka teorijski nećemo razmatrati, osim u praktičnim primerima kasnije. Više o ovim servisima možete pročitati u udžbeniku Kriptologija 1.

### 2.5.1 UPRAVLJANJE KLJUČEVIMA

Najveći problem kod simetričnih šifri je sistem za upravljanje ključevima. Pod pojmom upravljanje ključevima podrazumevamo nekoliko elemenata koji su podjednako važni. Suština je, izgradnja kriptografskog sistema preko koga će dve

strane šifrovano da komuniciraju. Treba imati na umu da su podaci apsolutno bezbedni sve dok je lozinka ili ključ u glavi jedne osobe, ali u momentu kada ta osoba kaže lozinku drugoj osobi ili je zapiše negde, tada se otvaraju novi problemi, kao što je problem, generisanja, skladištenja ili problem distribucije ključa između dve strane, koji zahteva bezbedan kanal preko koga je moguće organizovati razmenu ključeva.

U narednom poglavlju diskutovaćemo o mogućnosti da upotrebimo asimetrične šifre u protokolima za bezbednu distribuciju simetričnog ključa.

## 2.6 SIMETRIČNE ŠIFRE - ŠIFARSKA KLASA

`Cipher` klasa predstavlja zajedničku osnovu svih algoritama dostupnih u *JCE* ekstenziji. Radi kreiranja instance `Cipher`, pozivamo metod `getInstance()`, zatim prosleđujemo argument koji predstavlja naziv date transformacije. Opciono je moguće definisati naziv provajdera koji će se koristiti.

Transformacija je string koji predstavlja operaciju (ili paletu transformacija) koje će biti korišćene nad ulaznim podacima (poruka), da bi kao rezultat generisala izlazni podatak (šifrat).

Sledeći kod je primer kreiranja instance iz klase `Cipher`:

```
Cipher desSifra = Cipher.getInstance("DES");
```

Navedeni fragment koda konstruisaće naizgled samo instancu *DES* šifre, ali pitanje je koji šifarski režim i koja šema dopunjavanja se koristi, obzirom da nije navedeno. Prema specifikaciji Java *JCE* ekstenzije, kada nisu eksplicitno navedeni režimi za šifrovanje i šeme dopunjavanja, tada je izbor prepušten provajderu. Na osnovu ovoga možemo da zaključimo da nam *JCE* daje mnogo ili malo kontrole, u zavisnosti koliko mi to želimo. U ovom slučaju mi smo se odlučili za manje kontrole, zahtevali smo instancu *DES* algoritma, a odustali smo od samostalnog izbora šifarskog režima i šeme dopunjavanja.

Treba imati na umu da svaki krajnji korisnik može da menja sadržaj datoteke `java.security` na svom računaru i iz tog razloga ne treba praviti pretpostavku da će

određeni provajder uvek biti prvi na listi. U nastavku sledi standardna lista definisanih provajdera na računaru korisnika:

```
1. security.provider.1=sun.security.provider.Sun
2. security.provider.2=sun.security.rsa.SunRsaSign
3. security.provider.3=sun.security.ec.SunEC
4. security.provider.4=com.sun.net.ssl.internal.ssl.Provider
5. security.provider.5=com.sun.crypto.provider.SunJCE
6. security.provider.6=sun.security.jgss.SunProvider
7. security.provider.7=com.sun.security.sasl.Provider
8. security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
9. security.provider.9=sun.security.smartcardio.SunPCSC
10. security.provider.10=sun.security.mscapi.SunMSCAPI
```

U prethodnom primeru Java koda nije nam poznato koji je provajder korišćen, koji je šifarski režim primenjen i koja je šema dopunjavanja primenjena. Sun*JCE* provajder eksplicitno dokumentuje da *DES*, *DES-EDE* i *Blowfish* šifra koriste inicijalno *ECB* režim za šifrovanje i *PKCS#5* šemu dopunjavanja.

U nastavku poglavlja diskutovaćemo više o režimima šifrovanja i šemama dopune, ukoliko je potrebno više teorijske osnove za režime šifrovanja, pronaći u udžbeniku „Kriptologija 1“.

U praksi može da se dogodi da koristimo neki drugi provajder koji podržava *DES* šifru i da je njegov inicijalni režim šifrovanja *CBC* (situacija u kojoj je modifikovana `java.security` datoteka), a da druga strana koristi provajder koji standardno podržava *ECB*. U tom slučaj dobićemo šifrovanje sa režimom *CBC* i dešifrovanje sa režimom *ECB*, a to znači da nećemo dobiti originalnu poruku prilikom dešifrovanja.

Međutim, ukoliko se odigrava pun ciklus kriptografskog procesa (šifrovanje i dešifrovanje), treba imati uvek na umu da podešenost *JVM* (Java virtuelna mašina) na svakom računaru može da bude različita.

Prevazilaženje ovakvog problema je jednostavno, uvođenjem dodatnih argumenata u momentu kreiranja instance za izabranu šifru. `Cipher` klasa podržava eksplicitno navođenje režima za šifrovanje i šeme za dopunjavanje.

Da bismo izbegli situacije o kojima smo diskutovali, predlažemo upotrebu argumenta sledećeg formata:

DES/CBC/PKCS5Padding

gde *DES* označava šifarski algoritam, *CBC* režim za šifrovanje i *PKCS5Padding* šemu dopunjavanja za otvoreni tekst, a svi parametri zajedno obaveštavaju `Cipher` klasu da samostalno želimo da napravimo izbor parametara.

Svaka implementacija Java platforme je u obavezi da obezbedi podršku za sledeće standardne `Cipher` transformacije sa veličinom ključa u zagradama:

1. AES/CBC/NoPadding (128)
2. AES/CBC/PKCS5Padding (128)
3. AES/ECB/NoPadding (128)
4. AES/ECB/PKCS5Padding (128)
5. DES/CBC/NoPadding (56)
6. DES/CBC/PKCS5Padding (56)
7. DES/ECB/NoPadding (56)
8. DES/ECB/PKCS5Padding (56)
9. DESede/CBC/NoPadding (168)
10. DESede/CBC/PKCS5Padding (168)
11. DESede/ECB/NoPadding (168)
12. DESede/ECB/PKCS5Padding (168)
13. RSA/ECB/PKCS1Padding (1024, 2048)
14. RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)
15. RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

Ove transformacije su opisane u `Cipher` dokumentaciji *JCASAND* (*Java Cryptography Architecture Standard Algorithm Name Documentation*).

Sledeći kod je primer kreiranja instance iz klase `Cipher` sa navođenjem više parametara u vidu jednog argumenta:

```
Cipher desSifra = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

U ovom slučaju, ukoliko dođe do razlike u konfiguraciji *JVM*-e, kriptografski ciklus šifrovanje/dešifrovanje će biti uspešno obavljen, jer su jasno navedeni svi parametri.

Prethodni primer ukazuje na jedan značajan kriptografski koncept, kada se govori o tajnosti korišćenog algoritma. Korišćeni parametri nisu tajni, ali to ne znači da ih treba javno objavljivati. Čak, i ako neki kriptoanalitičar presretne šifrat i dozna konfiguracione parametre šifre, bezbednost će i dalje ostati na istom nivou. Svakako, kada govorimo o tim parametrima, ne mislimo na tajni simetrični ključ, za koji važi pretpostavka da je izuzetnog kvaliteta i da ostaje uvek u strogoj tajnosti između dve strane. Ne zaboravimo da se kriptografske operacije oslanjaju na čvrste matematičke principe, koji u ovom slučaju obezbeđuju računarski tip bezbednosti.

### 2.6.1 INICIJALIZOVANJE SIMETRIČNE ŠIFRE

Posle kreiranja šifarske instance preko klase `Cipher`, koji implementira zahtevanu šifru, šifarski režim i šemu dopunjavanja otvorenog testa, potreban je još jedan korak koji je obavezan. To je korak koji upravo određuje opšti režim šifre, konkretno mislimo na režim za šifrovanje ili dešifrovanje. Šifru nije moguće u istom trenutku koristiti za operaciju šifrovanja ili dešifrovanja. Moguće je koristiti samo onu operaciju koja je inicijalizovana preko preopterećenog metoda `init`().

Postoji više ovakvih preopterećenih metoda, koji raspolažu sa dva, tri i četiri argumenta. Jednim argumentom inicijalizujemo operaciju, dok drugim argumentima prosleđujemo parametre kao što su: tajni ključ ili specifikaciju ključa ili izvor slučajnosti ili sertifikati.

Obzirom da imamo mogućnost da preko `init`() metoda promenimo opšti režim rada šifre (šifruj/dešifruj), bitno je imati na umu da će novim pozivom `init()` metoda šifra biti novo inicijalizovana. Tačnije, imaćemo potpuno novoizgrađeni objekat šifre.

Na primer, ukoliko imate više podataka koje želite da šifrujete različitim ključevima, tada je bitno svaki put koristiti metod `init()` za inicijalizovanje novog tajnog ključa.

Postoji slučaj kada `init()` metoda ne može da obezbedi sve potrebne parametre za šifru. Taj problem može da se javi u režimu dešifrovanja, ako je u fazi šifrovanja

korišćen inicijalni vektor (IV), a u procesu dešifrovanja nije obezbeđen identičan IV, tada rezultat dešifrovanja nije otvoreni tekst.

*Tabela 2.5 Lista preopterećenih metoda `init()` u klasi `Cipher`.*

| Tip rezult | Metod i opis metoda |
|---|---|
| **void** | **init**(int operacija, **Certificate** sertifikat)<br><br>Inicijalizuje šifru sa javnim ključem za dati sertifikat. |
| **void** | **init**(int operacija, **Certificate** sertifikat, **SecureRandom** izvorSlučajnosti)<br><br>Inicijalizuje šifru sa javnim ključem i izvorom slučajnosti. |
| **void** | **init**(int operacija, **Key** ključ)<br><br>Inicijalizuje šifru sa ključem. |
| **void** | **init**(int operacija, **Key** ključ, **AlgorithmParameters** parametri)<br><br>Inicijalizuje šifru sa ključem i setom parametara od korišćenog algoritma. |
| **void** | **init**(int operacija, **Key** ključ, **AlgorithmParameterSpec** parametri)<br><br>Inicijalizuje šifru sa ključem i specifikacijom parametara algoritma. |
| **void** | **init**(int operacija, **Key** ključ, **AlgorithmParameterSpec** parametri, **SecureRandom** izvorSlučajnosti)<br><br>Inicijalizuje šifru sa ključem, specifikacijom parametara algoritma i izvorom slučajnosti. |

| | |
|---|---|
| **void** | **init**(int operacija, **Key** ključ, **AlgorithmParameters** parametri, **SecureRandom** izvorSlučajnosti)<br><br>Inicijalizuje šifru ključem, setom algoritamskih parametara i izvorom slučajnosti. |
| **void** | **init**(int operacija, **Key** ključ, **SecureRandom** izvorSlučajnosti)<br><br>Inicijalizuje šifru ključem i izvorom slučajnosti. |

### 2.6.2 ŠIFROVANJE BLOKOVSKOM ŠIFROM

Obzirom da postoji više `init()` preopterećenih metoda (Tabela 2.5), koristićemo u primeru one čija je upotreba najčešća ili koje prevladavaju za većinu simetričnih algoritama. Sledi potpis metoda koji ćemo da koristimo u ovom primeru:

```
public void init(int operacija, Key ključ);
```

Argument `operacija` predstavlja direktivu za operaciju (šifrovanje / dešifrovanje) koja će biti primenjena. Za simetrične šifre koriste se statičke konstante `Cipher.ENCRYPT_MODE` ili `Cipher.DECRYPT_MODE`, umesto literala celobrojne `INT` vrednosti. Argument `Key` predstavlja tajni ključ koji je ranije generisan. Šifra će koristiti ovaj ključ da obavi inicijalizovanu operaciju.

U zavisnosti od šifarskog režima i šeme dopune otvorenog teksta, klasa `Cipher` može dinamički da generiše inicijalni vektor *IV*. Za pristup korišćenom inicijalnom vektoru, moguće je pristupiti preko javnog metoda interfejsa `getIV()`.

U nastavku, preći ćemo na primer (Primer 2.4) u kome ćemo prvo generisati tajni ključ, inicijalizovati AES šifru i obaviti operaciju šifrovanja. Pored osnovnih funkcija, koristićemo i druge metode za dobavljanje više detalja o kompletnom kriptografskom postupku.

Primer šifrovanja sa *AES/CBC/PSCS5Padding* specifikacijom algoritma, korišćen je za šifrovanje jednog seta podataka. Ovo je vrlo praktično ukoliko se obavlja samo jedno šifrovanje.

Međutim, u praksi mogu da postoje i drugi zahtevi, a to je situacija u kojoj imamo da šifrujemo više setova podataka koji možda dolaze iz različitih izvora sa istim parametrima šifre. U tom slučaju, šifarski algoritam mora da ostane u radnom stanju da bi pravilno obavio kompletan kriptografski proces za operaciju šifrovanja.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

```
public class AESsifra {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoS
        //Generisanje ključa
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        SecretKey kljuc = kg.generateKey();
        byte[] AESkljuc = kljuc.getEncoded();
        SecretKeySpec specifikacijaKljuca = new SecretKeySpec(AESkljuc, "AES");
        //Inicijalizovanje šifre
        Cipher sifra = Cipher.getInstance("AES/CBC/PKCS5Padding");
        sifra.init(Cipher.ENCRYPT_MODE, specifikacijaKljuca);
        //Poruka za šifrovanje
        String poruka = "UNIVERZITET SINGIDUNUM 2015.";
        //Šifrovanje poruke
        byte sifrat[] = sifra.doFinal(poruka.getBytes("UTF-8"));

        System.out.println("Algoritam kljuca: "+kljuc.getAlgorithm());
        System.out.println("Veličina ključa: "+kljuc.getEncoded().length*8);
        System.out.println("Veličina bloka: "+sifra.getBlockSize());
        System.out.println("Šifarski algoritam: "+sifra.getAlgorithm());
        System.out.println("Provajder: "+sifra.getProvider());
        System.out.println("IV: "+new String(sifra.getIV()));
        System.out.println("Poruka: "+poruka);
        System.out.println("Sifrat: "+new String(sifrat));
    }
}
```

*Primer 2.4* AES *šifrovanje - AES/CBC/PSCS5Padding*

Programski izlaz:

```
Algoritam kljuca: AES
Veličina ključa: 128
Veličina bloka: 16
Šifarski algoritam: AES/CBC/PKCS5Padding
Provajder: SunJCE version 1.8
IV: F}□$* □V□□,mw□□&
Poruka: UNIVERZITET SINGIDUNUM 2015.
Sifrat: □□p  □□ □  □□□tÿ□S9□L□ D□□□
```

Metod koji ćemo iskoristi u sledećem primeru (Primer 2.7) je metod `update()` iz klase `Cipher`. Na kraju kada smo transformisali poruke u šifrat, šifrat ćemo upisati u fajl „Sifrat.AES“ koji se nalazi u *root*-u Java projekta. Metod `update()` predviđen je za obavljanje višedelnih operacija, situacija u kojoj nemamo kompletan sadržaj poruke za šifrovanje.

```
public class AESsifra2 {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //Inicijalizovanje tajnog ključa za AES algoritam
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        SecretKey kljuc = kg.generateKey();
        byte[] AESkljuc = kljuc.getEncoded();
        SecretKeySpec specifikacijaKljuca = new SecretKeySpec(AESkljuc, "AES");

        //Inicijalizovanje parametara AES algoritma - šifra
        Cipher sifra = Cipher.getInstance("AES/CBC/PKCS5Padding");
        sifra.init(Cipher.ENCRYPT_MODE, specifikacijaKljuca);

        //Poruke
        String poruka1 = "UNIVERZITET SINGIDUNUM 2015.";
        String poruka2 = "Fakultet za informatiku i računarstvo!";
        String poruka3 = "Praktikum iz zaštite informacionih sistema!";

        //Pozivanje update() metoda pojedinačno za svaku novu poruku
        sifra.update(poruka1.getBytes("UTF-8"));
        sifra.update(poruka2.getBytes("UTF-8"));
        sifra.update(poruka3.getBytes("UTF-8"));

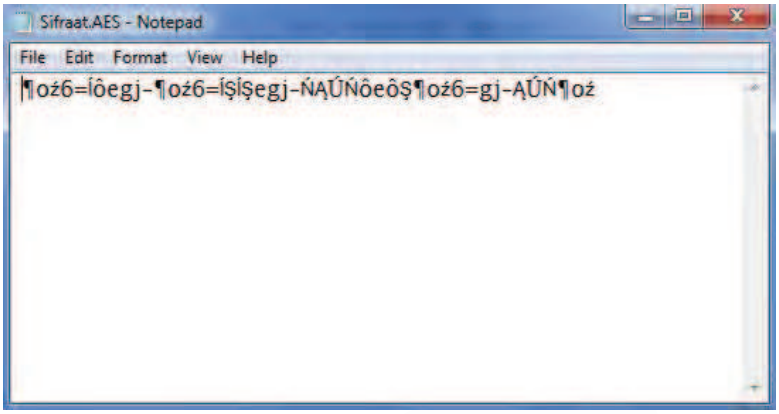
        //Šifrovanje - kompletiranje šifarske operacije
        byte sifrat[] = sifra.doFinal();

        //Upis šifrata u fajl
        Path lokacija = Paths.get("Sifraat.AES");
        Files.write(lokacija, sifrat, StandardOpenOption.CREATE_NEW);
    }
}
```

*Primer 2.5 AES šifrovanje sa korišćenjem metoda `update()` i `doFinal()`*

Obzirom da postoje mnoge kombinacije metoda `update()` i `doFinal()`, ipak većina blokovskih šifara ima tendenciju da koristi samo `doFinal()` bez metoda `update()`, jer korišćenjem oba metoda moramo da budemo spremni da obradimo izuzetke koji se odnose na šemu dopunjavanja otvorenog teksta (engl. *Bad Padding Exception*) i veličinu ulaznih blokova otvorenog teksta (engl. *Illegal Block Size Exception*).

Na sledećoj slici prikazan je sadržaj datoteke „Sifrat.AES“ koja u sebi sadrži šifrat dobijen preko prethodnog primera.




*Slika 2.6 AES šifrat*

### 2.6.3 ŠIFROVANJE U *CFB* REŽIMU – BLOKOVSKA ŠIFRA

Pričali smo već o konceptualnoj razlici između blokovskog i sekvencijalnog šifrovanja. Zapravo, možemo da kažemo da je sekvencijalna šifra blokovska šifra koja se izvršava na jednom bloku. Na primer, u jednom taktu šifruje se jedan bajt. Upravo takva operacija predstavlja trivijalan zadatak za *JCE* proširenje, čiji je zadatak da pretvori blok u tok podataka (konvertovanje dvodimenzionalnog u jednodimenzionalni podatak).

Nakon primene jednostavne funkcije za ovu transformaciju, sve dalje prepuštamo šifarskom režimu, tj. da definiše algoritam *CFB16* i broj bitova koji će šifrovati ili dešifrovati u jednom taktu ili vremenskom trenutku.

Kada smo podesili broj bitova koji šifra treba da šifruje u jednom taktu, tada lako možemo segmentirati ulazne podatke preko klase `JavaInputStream` ili `CipherInputStream`. Zapravo, argumenti konstruktora za instancu klase `CipherInputStream` su instanca `JavaInputStream` i instanca `Cipher`. Na ovaj način svi podaci koji se nalaze u toku podataka su šifrovani za operaciju šifrovanje ili dešifrovani za operaciju dešifrovanje. Ovo predstavlja bezbedan način za šifrovanje datoteka.

U primeru (Primer 2.6), iterativno ćemo učitati sadržaj fajla „moji_podaci.txt“ i upisati šifrat u novi fajl „moji_podaci_sifrat.aes“.

```
public class AESsifraCFB16 {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //Generisanje tajnog ključa
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        SecretKey kljuc = kg.generateKey();
        byte[] AESkljuc = kljuc.getEncoded();
        SecretKeySpec specifikacijaKljuca = new SecretKeySpec(AESkljuc, "AES");

        //CFB šifruje po 16 bitova u jednom taktu
        Cipher sifra = Cipher.getInstance("AES/CFB16/PKCS5Padding");
        sifra.init(Cipher.ENCRYPT_MODE, specifikacijaKljuca);
        FileInputStream fis = new FileInputStream("moji_podaci.txt");
        CipherInputStream cis = new CipherInputStream(fis, sifra);

        //Upis šifrata u datoteku
        File datoteka = new File("moji_podaci_sifrat.AES");
        FileOutputStream fos = new FileOutputStream(datoteka);

        //Broj bitova koji će biti prosesuiran u toku po jednom taktu
        byte cipherText[]= new byte[16];
        int i=0;
        while((i=cis.read(cipherText))!= -1){
            fos.write(cipherText,0,i);
        }
        System.out.println("Svi podaci iz toka su šifrovani!");

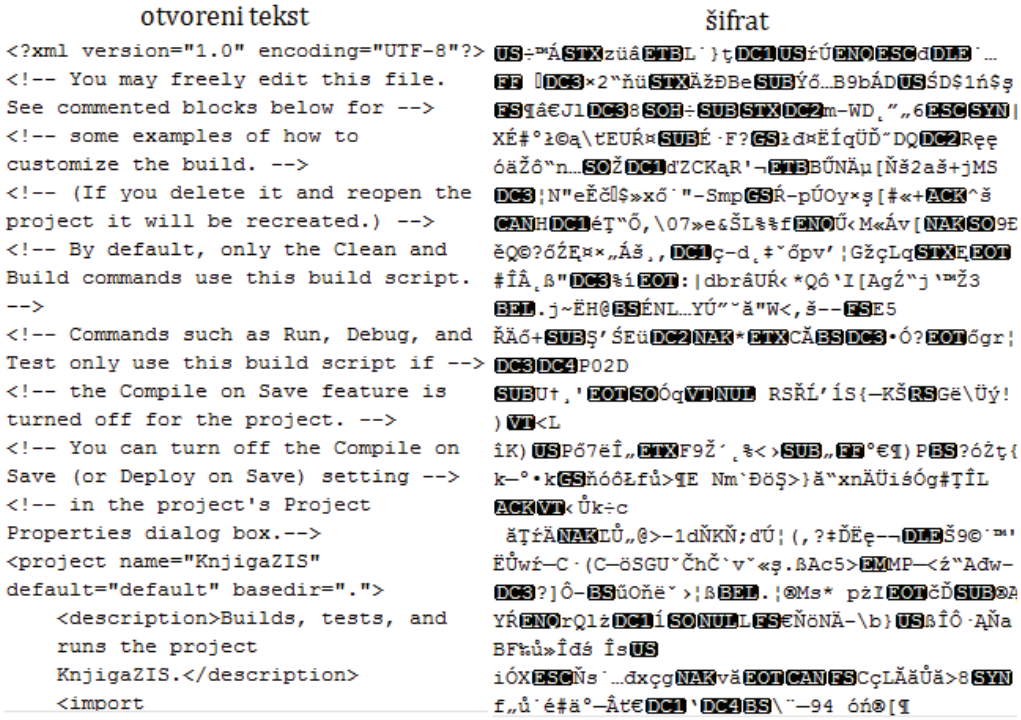
        //Memorisanje ključa i IV-a
        AESsifraCFB16.sacuvajIV(sifra.getIV(), "IV.txt");
        AESsifraCFB16.sacuvajKljuc(specifikacijaKljuca.getEncoded(),"kljuc.txt")
    }
    //Metode za upis ključa i IV u datoteku
    public static void sacuvajKljuc(byte kljuc[], String nazivDatoteke) throws I
        Path lokacija = Paths.get(nazivDatoteke);
        Files.write(lokacija, kljuc, StandardOpenOption.CREATE_NEW);
    }
    public static void sacuvajIV(byte IV[], String nazivDatoteke) throws IOExcep
        Path lokacija = Paths.get(nazivDatoteke);
        Files.write(lokacija, IV, StandardOpenOption.CREATE_NEW);
    }
}
```

*Primer 2.6 Šifrovanje sekvencijalnim šifarskim režimom* - CFB

Programski izlaz:

```
Svi podaci iz toka su šifrovani!
```

Na sledećoj slici prikazan je sadržaj fajla „moji_podaci.txt“ (otvoreni tekst) i sadržaj fajla „moji_podaci_sifrat.aes“ posle obavljenog šifrovanja sa algoritamskom specifikacijom *AES/CFB16/PKCS5Padding*.




*Slika 2.7 Rezultat* - AES/CFB16/PKCS5Padding

Ne zaboravimo da je uz režim povratne sprege *CFB* zahtevan i inicijalni vektor *IV*. Obzirom da nismo eksplicitno generisali ili prosledili inicijalni vektor, već je inicijalni vektor dinamički generisan preko `Cipher` klase. Iz tog razloga neophodno je preko metoda `getIV()` sačuvati inicijalni vektor, da bi u budućnosti operacija dešifrovanje bila moguća.

### 2.6.4 DEŠIFROVANJE U *CFB* REŽIMU – BLOKOVKSA ŠIFRA

Pre nego što pređemo na primer dešifrovanja sekvencijalnom šifrom, slede dva metoda preko kojih je moguće memorisati ključ i inicijalni vektor u fajl. Drugi još pouzdaniji način je serijalizacija objekta, čime bi bila memorisana sva stanja šifarske instance i specifikacija tajnog ključa.

Pod pojmom serijalizacije praktično podrazumevamo zamrzavanje svih stanja objekta i memorisanje istog u bajt formatu. Na ovaj način sigurni smo da neće doći do promene u svim parametrima u kasnijoj upotrebi. U svakom momentu treba imati na umu da je kriptografija nauka koja očekuje apsolutnu tačnost bez i malo odstupanja. U protivnom možemo da trajno izgubimo podatke.

```
public static void sacuvajKljuc(byte kljuc[], String nazivDatoteke) throws IOException{
    Path lokacija = Paths.get(nazivDatoteke);
    Files.write(lokacija, kljuc, StandardOpenOption.CREATE_NEW);
}
public static void sacuvajIV(byte IV[], String nazivDatoteke) throws IOException{
    Path lokacija = Paths.get(nazivDatoteke);
    Files.write(lokacija, IV, StandardOpenOption.CREATE_NEW);
}
```

*Slika 2.8 Metodi za zapisivanje tajnog ključa i inicijalnog vektora u fajl*

U primeru (Primer 2.6), potrebno je dodatno pozvati ova dva metoda, da bismo obezbedili tajni ključ i inicijalni vektor *IV* za upotrebu u primeru (Primer 2.9).

```
AESsifraCFB16.sacuvajIV(sifra.getIV(), "IV.txt");
AESsifraCFB16.sacuvajKljuc(specifikacijaKljuca.getEncoded(), "kljuc.txt");
```

*Slika 2.9 Zapisivanje tajnog ključa i inicijalnog vektora* IV

Da bismo uspešno obavili operaciju dešifrovanja čiji primer sledi, neophodno je učitati iz fajla tajni ključ i inicijalni vektor *IV*, koji smo prethodno memorisali u fajl „kljuc.txt“ i „IV.txt“. Sledeći metode (Slika 2.10) koristićemo za učitavanje ključa i *IV*.

```
public static byte[] ucitajKljuc(String nazivFajla) throws IOException{
    Path lokacija = Paths.get(nazivFajla);
    return Files.readAllBytes(lokacija);
}
public static byte[] ucitajIV(String nazivFajla) throws IOException{
    Path lokacija = Paths.get(nazivFajla);
    return  Files.readAllBytes(lokacija);
}
```

*Slika 2.10 Metode za učitavanje tajnog ključa i inicijalnog vektora* IV *iz datoteke*

```
public class AESsifraCFB16_D {
    public static void main(String[] args) throws IOException, NoSuchAlgorithmE>
        //Učitavanje tajnog ključa i inicijalnog vektora
        byte[] AESkljuc = AESsifraCFB16_D.ucitajKljuc("kljuc.txt");
        byte[] IV = AESsifraCFB16_D.ucitajIV("IV.txt");

        //Generisanje instanci za ključ, IV i šifru
        SecretKeySpec specifikacijaKljuca = new SecretKeySpec(AESkljuc, "AES");
        IvParameterSpec iv = new IvParameterSpec(IV);
        Cipher sifra = Cipher.getInstance("AES/CFB16/PKCS5Padding");
        sifra.init(Cipher.DECRYPT_MODE, specifikacijaKljuca, iv);

        //Proces dešifrovanja
        FileInputStream fis = new FileInputStream("moji_podaci_sifrat.AES");
        CipherInputStream cis = new CipherInputStream(fis, sifra);
        File datoteka = new File("moji_podaci_otvoreni_tekst.txt");
        FileOutputStream fos = new FileOutputStream(datoteka);

        //Broj bitova koji će biti procesuiran u toku po jednom taktu
        byte cipherText[]= new byte[16];
        int i=0;
        while((i=cis.read(cipherText))!= -1){
            fos.write(cipherText,0,i);
        }
        System.out.println("Svi podaci iz toka su dešifrovani!");
    }
    //Metodi za učitavanje tajnog ključa i IV
    public static byte[] ucitajKljuc(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
    public static byte[] ucitajIV(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return  Files.readAllBytes(lokacija);
    }
}
```

*Primer 2.7 Dešifrovanje sekvencijalnim šifarskim režimom* - CFB

Programski izlaz:

```
Svi podaci iz toka su dešifrovani!
```

U primeru (Primer 2.7), prvo smo učitali bajtove ključa u *RAW* formatu, koji reprezentuju tajni ključ veličine 128 bitova i inicijalni vektor. Bajtove ključa u *RAW* formatu konvertovali smo u specifikaciju ključa preko klase `SecretKeySpec`. Ovaj način je dobar primer u kome smo direktno koristili `SecretKeySpec` klasu, jer je tajni ključ već jednom korišćen i ne dobijamo ništa na dodatnoj vrednosti upotrebom `SecretKeyFactory` klase.

## 2.7 ŠIFROVANJE NA OSNOVU LOZINKE - *PBE*

Upotreba lozinke u ulozi tajnog ključa za šifrovanje nije preporučljivo. Lozinke su generisane na osnovu prirode ljudske slučajnosti. Često, takve lozinke su kratke ili se lako pamte ili su fraze, a samim tim ne daju dovoljnu količinu informacije za generisanje jakog tajnog ključa. Umesto lozinki preporučujemo upotrebu kombinacije klasa `KeyGenerator` i `SecureRandom`.

U svakom slučaju, važno je napraviti dobru procenu u zavisnosti od potrebe aplikacije. Nekad je bolje koristiti *PBE* sistem (engl. *Password Based Encryption*), nego lozinke čuvati u instancama klase `String`. Za to postoji više razloga, dovoljno je reći da String instance nije moguće poništiti kada prestanemo koristiti lozinku. Zamena za `String` instancu je niz karaktera, koji treba uvek koristiti u radu sa lozinkama. Kada lozinka nije više potrebna, jednostavno možemo na poziciju svakog elementa u nizu postaviti vrednost *NULL*, ovim činom lozinka je zauvek izbrisana iz memorije.

Lozinke koje generišu ljudi su predvidljive ili najčešće neke fraze koje je mogu pronaći u rečniku. Aplikacije takve lozinke čuvaju u bazama podatak. Lozinke su memorisane u vidu heš vrednosti. Servisi za autentifikaciju proveru obavljaju tako što upoređuju dve heš vrednosti, ukoliko se poklapaju, lice koje se autentifikuje zna pravu lozinku. Međutim, jednostavno je organizovati hibridni napad navale, koji podrazumeva pretragu svih ključeva uz pomoć rečnika. Vidimo da u ovom slučaju heš ne predstavlja dovoljno dobro rešenje.

Za rešavanje ovakvog problema potrebno je uvesti neke dodatne vrednosti. *Salt* (engl. *Salt*) predstavlja slučajno izabranu vrednost koja se dodaje na lozinku. Na ovaj način se značajno uvećava prostor ključeva i sprečava napad koji smo malopre pomenuli, ali i dalje ga ne čini neprobojnim. Bitno je razumeti da dodavanje ove vrednosti ne poboljšava kvalitet lozinki koje su smislili ljudi.

U nastavku sledi primer koda u kome koristimo takve lozinke sa uvođenjem dodatnih *salt* vrednosti i heš funkcija za primenu u simetričnim šiframa.

### 2.7.1 ŠIFROVANJE NA OSNOVU LOZINKE – BLOKOVSKA ŠIFRA

U poglavlju 2, već smo se susreli sa kodovima za pretvaranje ključa prema specifikaciji ključa za određeni algoritam i diskutovali o prednostima. Posebno, u primeru kada smo koristili klasu `DESKeySpec` u kombinaciji sa klasom `SecretKeyFactroy`. Takođe, već smo govorili i o problemu nepromenljivosti, koji se odnosi na String instance u slučaju šifrovanja sa lozinkama – *PBE*.

Sada uvodimo u upotrebu jednu novu klasu za rad sa *PBE* lozinkama. Klasa `PBEKeySpec` u paketu `javax.crypto.spec`, definiše sve funkcije za nizove karaktera i String instance.

Šifrovanje na osnovu lozinki čvrsto se oslanja na dodatnu *salt* vrednost i zadati broj iteracija mešanja. Uloga mešanja je da dodatno pomuti reči definisane u rečniku i da dodatno zaštiti lozinku. Preporučeni broj iteracija je 1000. Pristup *salt* vrednosti i lozinci možemo kasnije obezbediti preko klase `PBEKeySpec`, jer se radi o specifikaciji ključa koja je po definiciji je transparentna.

Nijedna od ove dve informacije ne predstavlja tajnu vrednost i obe mogu da se dele između strane A i strane B, bez ugrožavanje snage tajnog ključa. Sledeći fragment koda predstavlja deklaraciju od 8 bajtova *salt* vrednosti, broj iteracija, lozinku i konstrukciju instance klase `PBEKeySpec.`

```
//Salt vrednost, 8 bajtova
byte[] saltVrednost =
        new byte[] { (byte)0xc0, (byte)0xd2, (byte)0x63, (byte)0x4a,
                     (byte)0x49, (byte)0x2a, (byte)0x98, (byte)0x37 };
//Broj iteracija
int iteracija = 1000;
//Unos lozinke
System.out.println("Unesite lozinku:");
char lozinka[] = new Scanner(System.in).nextLine().toCharArray();
//PBE specifikacija ključa
PBEKeySpec pbeKljucSpec = new PBEKeySpec(lozinka,saltVrednost,iteracija);
SecretKeyFactory fabrika = SecretKeyFactory.getInstance("PBEWithMD5AndTripleDES");
SecretKey kljuc = fabrika.generateSecret(pbeKljucSpec);
```

*Slika 2.11* PBE *specifikacija ključa za* - PBEWithMD5AndTripleDES

Specifikacija *PBEWithMD5AndTripleDES* predstavlja *3DES* algoritam koji koristi 112 bitova veličinu ključa i *MD5* heš vrednost koju mnogi smatraju nebezbednom. Pored specifikacije prikazane u ovom fragmentu koda dostupne su još *PBEWithSHA1AndRC2_40* i *PBEWithMD5AndDES*. Sledi kompletan primer sa šifrovanjem koji implementira specifikaciju *PBEWithMD5AndDES*. Takođe, *DES* šifra ograničava *salt* vrednost na veličinu od 8 bajtova.

```
public class PBEtajniKljucTDES {
    public static void main(String[] args) throws UnsupportedEncodingException,

        //Salt vrednost, ograničena na 8 bajtova od strane DES algoritma
        byte[] saltVrednost =
                new byte[] { (byte)0xc0, (byte)0xd2, (byte)0x63, (byte)0x4a,
                             (byte)0x49, (byte)0x2a, (byte)0x98, (byte)0x37 };

        //Broj iteracija
        int iteracija = 1000;

        //Unos lozinke
        System.out.println("Unesite lozinku:");
        char lozinka[] = new Scanner(System.in).nextLine().toCharArray();

        //PBE specifikacija ključa
        PBEKeySpec pbeKljucSpec= new PBEKeySpec(lozinka,saltVrednost,iteracija);
        //Zahtevani set parametara - algoritam
        String specifikacija = "PBEWithHmacSHA1AndAES_128";
        SecretKeyFactory fabrika = SecretKeyFactory.getInstance(specifikacija);
        SecretKey kljuc = fabrika.generateSecret(pbeKljucSpec);

        //Inicijalizacija šifre
        Cipher sifra = Cipher.getInstance(specifikacija);
        sifra.init (Cipher.ENCRYPT_MODE, kljuc);

        System.out.println("Unesite poruku za šifrovanje:");
        byte poruka[] = new Scanner(System.in).nextLine().getBytes("UTF-8");

        //Šifrovanje unešene poruke
        byte sifrat[] = sifra.doFinal(poruka);
        System.out.println("Šifrat: "+new String(sifrat));
    }
}
```

*Primer 2.8 Šifrovanje po specifikaciji* - PBEWithMD5AndDES

Programski izlaz:

```
Unesite lozinku:
Singidunum 2@!5
Unesite poruku za šifrovanje:
Fakultet za informatiku i računarstvo
Šifrat: I : ~□z□x i □□Y□owjskR+š□U'\{J□M□ □x□>? c□□Q□
```

U primeru (Primer 2.8), pokazali smo izgradnju tajnog ključa na osnovu lozinke. Na početku koda definisali smo niz slučajno izabranih bajtova. Ukupan broj bajtova je 8.

Zatim, pokazali smo kako da preko klase `SecretKeyFactory` kreiramo instancu klase `SecretKey`. Konačno, kada smo generisali tajni ključ, izabrali šifarski algoritam i inicijalizovali isti, tada možemo da primenimo operaciju šifrovanja ili dešifrovanja.

Sledi lista dostupnih algoritama - *PBE*:

1. PBE
2. PBES2
3. PBEWithHmacSHA1AndAES_128
4. PBEWithHmacSHA224AndAES_128
5. PBEWithHmacSHA256AndAES_128
6. PBEWithHmacSHA384AndAES_128
7. PBEWithHmacSHA512AndAES_128
8. PBEWithHmacSHA1AndAES_256
9. PBEWithHmacSHA224AndAES_256
10. PBEWithHmacSHA256AndAES_256
11. PBEWithHmacSHA384AndAES_256
12. PBEWithHmacSHA512AndAES_256
13. PBEWithMD5AndDES
14. PBEWithMD5AndTripleDES
15. PBEWithSHA1AndDESede
16. PBEWithSHA1AndRC2_40
17. PBEWithSHA1AndRC2_128
18. PBEWithSHA1AndRC4_40
19. PBEWithSHA1AndPC4_128

## 2.8 ŠIFROVANJE SA SEKVENCIJALNOM ŠIFROM *RC4*

*RC4* je sekvencijalna šifra koja se u svojoj osnovi razlikuje od *A5/1*. Algoritam *RC4* je napravljen za softversku implementaciju, dok je *A5/1* dizajniran za hardver. *RC4* generiše bajt radnog ključa u jednom taktu, dok *A5/1* proizvodi u jednom taktu samo jedan bit radnog ključa. Sa aspekta vremenskih performansi, generisanje bajta u jednom taktu je mnogo bolje od generisanja jednog bita.

*RC4* algoritam je veoma jednostavan zato što je njegova osnova tabela koja sadrži permutaciju svih mogućih vrednosti od 0-255 bajtova. Suština je u sledećem: svaki put kada se generiše bajt radnog ključa, tabela se modifikuje na takav način da uvek sadrži permutaciju od {0,1,…,255}. Zbog ovog stalnog menjanja ili samo-modifikovanja, tabela i sama šifra za kriptoanalitičara predstavlja pokretnu metu.

Jedna interesantna odlika *RC4* algoritma jeste da ključ može biti bilo koje dužine od 1 do 256 bajtova. Najčešća dužina ključa je između 5 i 40 bajta. Ključ se koristi samo da bi se inicijalizovala permutacija *S*. Možemo da zapazimo da se 256 bajt-ni niz *K* dobija prosto ponavljanjem ključa dok se niz ne dovoljno ne napuni.

RC4 algoritam se može posmatrati kao samo promenljiva tabela. Vrlo je sofisticirano rešen, jednostavan i softverski efikasan. Međutim, postoje napadi koji su izvodljivi, ali samo pod nekim uslovima. Ukoliko je algoritam implementiran na adekvatan način, napad neće biti lako izvodljiv, a to znači da je prvih 256 bajtova radnog ključa odbačeno.

*RC4* se danas koristi u mnogim aplikacijama, uključujući zaštitne protokole *SSL* i *WEP*. *Google* danas koristi *RC4* algoritam u *TLS* zaštitnom protokolu za obezbeđivanje sigurne komunikacije između klijenta i servera u servisu za elektronsku poštu (*gmail*.*com*). U današnje vreme, u pružanju ovakvih usluga je neophodno koristiti kriptografsku zaštitu. Ne zaboravimo da je i *RC4* šifra pseudo *One-time pad* šifra i da su njene visoke performanse stvorile konkurentsku prednost nad ostalim modernim šiframa. Zbog ogromnog broja Internet korisnika i zahtevane brzine odziva, *Google* je izabrao *RC4* algoritam zbog svih svojih dobrih karakteristika.

### 2.8.1 IMPLEMENTACIJA *RC4 ŠIFRE*

U ovom delu prikazaćemo jednu jednostavnu implementacija *RC4* algoritma u Java programskom jeziku. Standardna veličina tajnog ključa je 128 bitova.

```
public class RC4 {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //Generisanje unutrašnjeg ključa - tajni ključ (RC4)
        KeyGenerator kg = KeyGenerator.getInstance("ARCFOUR");
        SecretKey kljuc = kg.generateKey();

        //Inicijalizovanje šifre
        Cipher sifra = Cipher.getInstance("ARCFOUR");
        sifra.init(Cipher.ENCRYPT_MODE, kljuc);

        //Unos poruke za šifrrovanje
        System.out.println("Unesite poruku za šifrovanje:");
        byte poruka[] = new Scanner(System.in).nextLine().getBytes("UTF-8");

        //Šifrovanje poruke
        byte sifrat[] = sifra.doFinal(poruka);
        System.out.println("Šifrat: "+new String(sifrat));
    }
}
```

*Primer 2.9 Java implementacija* RC4 *sekvencijalne šifre*

Programski izlaz:

```
Unesite poruku za šifrovanje:
Univerzitet Singidunum, Danijelova 32
Šifrat: □□ [.□□□ □ □}g□5□ ~□ ¨□=v□JqwF□t□□□}□
```

Sekvencijalne šifre su nekada bile primarne, ali su sada relativno retke u poređenju sa blokovskim šiframa. Neki su otišli toliko daleko da su proglasili i njihovo iščeznuće i kao dokaz naveli su da u poslednjih par godina nije bilo napora da se razvije ni jedna nova sekvencijalna šifra. Ipak, danas se javlja povećan broj značajnih aplikacija za koje su sekvencijalne šifre mnogo pogodnije nego blokovske. Primer za takve aplikacije su bežični uređaji (bežične mreže), uređaji sa ograničenom procesorskom snagom i uređaji sa velikim protokom podataka (zaštita videa i govora). Nesumnjivo, priče o iščeznuću blokovskih šifri su bile značajno preuveličane.

Na kraju ovog poglavlja, u vidu manjeg zaključka navešćemo po tačkama od čega se sastoji jedna Java implementacija kompletnog kriptografskog ciklusa:

- Generisanje tajnog ključa preko *CSPRNG* ili opciono koristimo lozinku kao niz karaktera;
- Na osnovu tajnog ključa konstruišemo specifikaciju ključa preko klase `SecretKeyFactory` ili alternativno preko klase `SecretKeySpec`;
- Konstruišemo instancu algoritma šifre preko klase Cipher;
- Inicijalizujemo instancu šifre za određeni radni režim (šifrovanje ili dešifrovanje);
- Memorišemo ili dalje šaljemo dobijeni šifrat ili otvorenu poruku;
- Iz bezbednosnih razloga brišemo na pouzdan način sve promenljive polazeći od *SCPRNG* pseudo izvora slučajnosti i korisnički definisanih lozinki *PBE*;

UNIVERZITET PRIVATNI AKREDITOVANI SINGIDUNUM

# 3. ASIMETRIČNI KLJUČEVI I IMPLEMENTACIJA ASIMETRIČNIH ŠIFRI

Asimetrični šifarski sistemi predstavljaju jedno od najvećih dostignuća kriptologije sedamdesetih godina dvadesetog veka. Nastali su na osnovu rešavanja ključnih problema simetričnih šifarskih sistema koji se odnose na distribuciju tajnih simetričnih ključeva. U literaturi se koriste i termini: kriptografija sa javnim ključevima (engl. *Public Key Cryprography – PKI*) ili kriptografija sa dva ključa (engl. *Key pair cryptography*).

Asimetrične šifre su našle svoju primenu u distribuiranoj arhitekturi. Kriptografija javnog ključa je često korišćeni termin koji opisuje asimetrične ključeve. Dizajn danas korišćenih protokola za razmenu tajnog ključa, zasnovan je na asimetričnim šiframa. Asimetrične šifre koriste par dva ključa za kompletan kriptografski postupak. Naziv jednog ključa je javni ključ, a drugog privatni ključ.

Ovi ključevi se koriste u različitim kriptografskim operacijama, kao što je digitalni potpis, operacija šifrovanja/dešifrovanja i u protokolima za razmenu tajnih ključeva. Treba imati na umu da svi algoritmi u asimetričnoj kriptografiji ne podržavaju sve operacije. Par ključeva za *DSA* algoritam se može jedino koristiti za digitalni potpis. Neke asimetrične šifre koriste parove ključeva koji su upotrebljivi u oba smera, tj. privatni ključ šifruje, a javni ključ dešifruje ili javni ključ šifruje, a privatni dešifruje.

Ključevi u asimetričnoj šifri su veoma kritična tačka. Napomenućemo, da je javni ključ namenjen za široku distribuciju, dok je privatni ključ tajan i od njega zavisi bezbednost digitalnog potpisa i šifrovanja podataka. U slučaju digitalnog potpisa, privatni ključ se koristi za generisanje digitalnog potpisa, dok je uloga javnog ključa u procesu verifikacije autentičnosti osobe koja je digitalno potpisala.

*PKI* sistemi se zasnivaju na matematičkim jednosmernim funkcijama sa zamkom (engl. *trap door one-way function*). To su funkcije koje se lako računaju u jednom smeru, a teško u drugom (računarski, tj. praktično neizvodljivo). Zamka osigurava da napadač ne može lako (direktno) da izračuna iz javnog ključa privatni ključ. Faktorizacija je relevantan primer za jednosmernu funkciju. Na osnovu dva prosta broja $p$ i $q$ lako je izračunati njihov proizvod $N = pq$, ali je za dovoljno veliki broj $N$, teško izračunati faktore $p$ i $q$.

Osobina asimetričnih ključeva je da ne govore jedan o drugom. Tačnije, ako je napadaču poznat javni ključ, tada on ne može da sazna bilo šta o privatnom ključu, dok su sa druge strane ključevi komplementarni i obavljanje kompletnog kriptografskog ciklusa zavisi od jednog određenog para asimetričnih ključeva.

Asimetrični šifarski sistemi imaju takve osobine da na jednostavan način obezbeđuju širi skup funkcija bezbednosti u odnosu na simetrične šifarske sisteme. Oni se karakteristično koriste za efikasnu autentifikaciju, integritet poruka, neporecivost i sl., a retko kada se koriste za funkciju poverljivosti. Najčešće se kombinuju sa simetričnim šifarskim sistemima. Na primer, jedna od popularnih primena PKI sistema je uspostavljanje simetričnog ključa za šifrovanje i dešifrovanje u simetričnom šifarskom sistemu.

Za razumevanje postupka implementacije asimetričnih šifri, neophodno je razumevanje materije iz prethodnog poglavlja, jer se *JCE* ekstenzija i dalje oslanja na klasu `Cipher`, o kojoj je do sad bilo puno reči.

## 3.1 GENERISANJE ASIMETRIČNIH KLJUČEVA

U ovom delu diskutovaćemo o ispravnom načinu za generisanje para asimetričnih ključeva. Svi asimetrični algoritmi zahtevaju par ključeva pre izvođenja kriptografske operacije. Odgovornost za generisanje kriptografski kvalitetnih ključeva, poverena je Java klasi `KeyPairGenerator` koja se nalazi u paketu `java.security`.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA") ;
```

Generator ključeva za određeni algoritam (navedeno u fragmentu koda iznad) generiše jedan javni i jedan privatni ključ, koji su prilagođeni za upotrebu sa traženim šifarskim algoritmom.

Postoje dva načina za generisanje para ključeva, jedina razlika je u inicijalizovanju objekta. Prvi način je algoritamski nezavisna inicijalizacija, a drugi inicijalizacija za tačno određeni algoritam. Ovo je sličan princip koji postoji kod inicijalizovanja generatora simetričnih ključeva, gde se koristi `init()` metoda.

Sve implementacije Java platforme zahtevaju podršku za sledeće algoritme i veličine ključeva unutar `KeyPaigGenerator` klase:

- Diffie-Hellman (1024) – generiše par ključeva za DH protokol ;
- DSA (1024) – generiše par ključeva za algoritam digitalnog potpisa;
- RSA (1024, 2048) – generiše par ključeva za digitalni potpis i šifrovanje;
- EC – generiše par ključeva za algoritam eliptične krive;

Pozivanjem metoda `getAlgoritam()` nad instancom ove klase, dobićemo standardni naziv jednog od gore navedenih algoritama. Navedeni algoritmi su opisani u sekciji `KeyPairGenerator`, u dokumentu Java kriptografska arhitektura i standardni nazivi algoritama.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

### 3.1.1 GENERISANJE KLJUČEVA NEZAVISNO OD ALGORITMA

Generisanje para ključeva preko klase `KeyPairGenerator` zahteva najmanje dve informacije, bez obzira na tip algoritma (veličina ključa u bitovima) i izvor slučajnosti. U primeru (Primer 3.1), pokazaćemo upotrebu klase za generisanje para asimetričnih ključeva.

```
import java.io.IOException;
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public class GenerisanjeParaKljuceva1 {

    public static void main(String[] args) throws NoSuchAlgorithmException, IOE
        //Generisanje para kljuceva nezavisnih od algoritma
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        //Inicijalizovanje veličine ključa
        kpg.initialize(1024);
        KeyPair parKljuceva = kpg.genKeyPair();
        //Zapisivanje javnog i privatnog ključa u datoteku
        System.out.println("Privatni ključ: \n"
                +new BigInteger(parKljuceva.getPublic().getEncoded()));
        System.out.println("Javni ključ: \n"
                +new BigInteger(parKljuceva.getPrivate().getEncoded()));
    }

}
```

*Primer 3.1 Generisanje asimetričnih ključeva*

Programski izlaz:

```
Privatni ključ:
25849353198772181885443536566619978312109721286452657611495574405087324632671070
57901986022565806048603725911492173817823133174927944097299971546607255560730014
29419100950748556767242599145936837845911424385733955596645051880049996664169412 7
05763992113969573442033734717970759244302033414529699208985931331744435421727440
96668386039514191248836295355415962516926777100276751372066413326499865
```




Slede potpisi preopterećene metode `initialize()`, koja je dostupne preko instance klase `KeyPairGenerator`:

1. `initialize(int)`
2. `initialize(int, SecureRandomm)`
3. `initialize(AlgorithmParameterSpec)`
4. `initialize(AlgorithmParameterSpec, SecureRandom)`

Prvi metod inicijalizuje veličinu ključa, koristeći standardne parametre i `SecureRandom` (izvor slučajnosti) implementaciju instaliranog provajdera sa najvećim prioritetom. Drugi metod inicijalizuje veličinu ključa i izvor slučajnosti na osnovu koga treba da se generišu ključevi. Treći metod inicijalizuje posebnu specifikaciju parametara i izvor slučajnosti preko provajdera sa najvećim prioritetom. Četvrti metod inicijalizuje posebnu specifikaciju parametara i izvor slučajnosti na osnovu koga treba da se generišu ključevi.

Pored ovih metoda tu su još metod `getPrivate()` i `getPublic()`, preko kojih uzimamo privatni i javni ključ. Kada su ključevi generisani, možemo ih koristiti za digitalni potpis. U zavisnosti od algoritma (ako podržava operacije), mogli bismo ih koristiti i za šifrovanje.

Posle ovog primera, vidimo da postoji značajna razlika između simetričnih i asimetričnih ključeva. Simetrične šifre su projektovane da koriste male veličine ključeva. *DES* šifra koristi veličinu ključa 56 bitova ili 7 bajtova. U današnje vreme, nivo zaštite koji pruža ključ veličine 56 bitova nije prihvatljiv za upotrebu. Savremene šifre, kao što je *AES* algoritam, mogu da koristi ključ veličine 128, 192 i 256 bitova, dok asimetrične šifre koriste veličinu ključa 1024 ili 2048 bitova. Bez obzira, ukoliko zanemarimo veličinu ključeva *DES*-a ili *AES*-a, vrlo važna osobina ovih ključeva je da su generisani potpuno slučajno na osnovu nekog prirodnog izvora informacije.

### 3.1.2 KODNE ŠEME ZA KODOVANJE KLJUČEVA

Do sada smo videli u radu sa simetričnim ključevima, da je tajni ključ predstavljen nizom bajtova i da odvojeno od njega može da bude prisutan i inicijalni vektor *IV*, koji se koristi ili ne, u zavisnosti od izabranog šifarskog režima. Tajni ključ je jednostavno zapisan u nekoj datoteci na fajl sistemu u *RAW* formatu, bez bilo kakvog dodatnog algoritma za kodovanje (engl. *Encoding*).

Međutim, asimetrični ključevi nisu jednostavni kao simetrični ključevi, asimetrični ključevi uključuju mnogo više informacija. Da bi zadržali osobinu nezavisnosti od provajdera, koriste se standardni formati. Javni ključevi se oslanjaju na upotrebu *X.509* standarda za kodovanje, dok se privatni ključevi oslanjaju na upotrebu *PKCS#8* standarda za kodovanje.

U primeru (Primer 3.2), generisaćemo ključeve od nekoliko poznatih algoritama, uključujući *AES*, šifrovanje na osnovu lozinke – *PBE* i *RSA*. Prikazaćemo dostupne meta podatke preko interfejsa instance ključa. Metoda interfejs `getFormat()` za rezultat vraća format kodovanja, a metoda `getAlgoritam()` naziv algoritma.

```
public class MetaPodaciKljuca {
    public static void main(String[] args) throws NoSuchAlgorithmException, Inva
        //AES - meta podaci
        KeyGenerator kg = KeyGenerator.getInstance("AES") ;
        SecretKey AESkljuc = kg.generateKey() ;
        System.out.println("Algoritam ključa: " + AESkljuc.getAlgorithm());
        System.out.println("Format kodovanja: " + AESkljuc.getFormat());
        //PBE - tajni ključ na osnovu lozinke - meta podaci
        byte[] saltVrednost =
                new byte[] { (byte)0xc0, (byte)0xd2, (byte)0x63, (byte)0x4a,
                             (byte)0x49, (byte)0x2a, (byte)0x98, (byte)0x37 };
        int iteracija = 100;
        char[] lozinka = new char[] { 'P', 'R', 'A', 'K', 'T', 'I', 'K', 'M'};
        PBEKeySpec pbeKljucSpec= new PBEKeySpec(lozinka,saltVrednost,iteracija);
        String specifikacija = "PBEWithHmacSHA1AndAES_128";
        SecretKeyFactory fabrika = SecretKeyFactory.getInstance(specifikacija);
        SecretKey kljuc = fabrika.generateSecret(pbeKljucSpec);
        System.out.println("Algoritam ključa: " + kljuc.getAlgorithm());
        System.out.println("Format kodovanja: " + kljuc.getFormat());
        //RSA - meta podaci
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA") ;
        kpg.initialize(2048);
        KeyPair parKljuceva = kpg.generateKeyPair();
        PublicKey javniKljuc = parKljuceva.getPublic();
        System.out.println("Algoritam ključa: " + javniKljuc.getAlgorithm());
        System.out.println("Format kodovanja: " + javniKljuc.getFormat());
        PrivateKey privatniKljuc = parKljuceva.getPrivate();
        System.out.println("Algoritam ključa: " + privatniKljuc.getAlgorithm());
        System.out.println("Format kodovanja: " + privatniKljuc.getFormat());
    }
}
```

*Primer 3.2 Prikaz meta podataka za simetrične i asimetrične ključeve*

Programski izlaz:

```
Algoritam ključa: AES
Format kodovanja: RAW
Algoritam ključa: PBEWithHmacSHA1AndAES_128
Format kodovanja: RAW
Algoritam ključa: RSA
Format kodovanja: X.509
Algoritam ključa: RSA
Format kodovanja: PKCS#8
```

U ovom praktičnom primeru smo videli da *AES* i *PBE* algoritam ne koriste algoritam za kodvanje tajnih ključeva, već su tajni ključevi predstavljeni kao sirovi bajtovi.

Takođe, videli smo da se za asimetrične ključeve primenjuju različiti algoritmi za kodovanje (*PKCS#8* i *X.509*).

Važno je reći da *JCA* ekstenzija uključuje još dve implementacije, koje se odnose na specifične formate kodovanja javnog i privatnog ključa:

- `java.security.spec.PKCS8EncodedKeySpec`
- `java.security.spec.X509EncodedKeySpec`

`PKCS8EncodedKeySpec` klasa se koristi za učitavanje privatnog ključa kodovanog sa *PKCS#8*, dok `X509EncodedKeySpec` klasa se koristi za javne ključeve kodovane preko *X.509* kodne šeme. Pozivom metode `getEncoded()` nad instancom ključa, metod za rezultat vraća niz bajtova za određeni tip ključa.

### 3.1.3 ZAPISIVANJE I UČITAVANJE ASIMETRIČNIH KLJUČEVA

U primeru (Primer 3.3), javni i privatni ključ generisani su preko klase `GenarateKeyPair`. Zatim, preko metode `sacuvajKljuc()`, upisaćemo ključeve u odvojene fajlove (JavniKljuc.JK i PrivatniKljuc.PK).

```
public class SacuvajJavniPrivatniKljuc {
    public static void main(String[] args) throws NoSuchAlgorithmException, IOEx
        //Generisanje javnog i prvatog ključa
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA") ;
        kpg.initialize(2048);
        KeyPair parKljuceva = kpg.generateKeyPair();
        PublicKey javniKljuc = parKljuceva.getPublic();
        PrivateKey privatniKljuc = parKljuceva.getPrivate();
        sacuvajKljuc(javniKljuc.getEncoded(), "JavniKljuc.JK");
        sacuvajKljuc(privatniKljuc.getEncoded(), "PrivatniKljuc.PK");
    }
    public static void sacuvajKljuc(byte kljuc[], String nazivFajla) throws IOEx
        Path lokacija = Paths.get(nazivFajla);
        Files.write(lokacija, kljuc, StandardOpenOption.CREATE_NEW);
    }
}
```

*Primer 3.3 Zapisivanje javnog i privatnog ključa u datoteku*

Programski izlaz:




U sledećem primeru, pokazaćemo kako možemo na ispravan način da opravimo ključeve (ponovo da ih učitamo u instancu ključa). U zavisnosti od ključa, koristićemo klase `PKCS8EncodedKeySpec` i `X509EncodedKeySpec`, zbog razlike u formatima (*X.509* i *PKCS#8*). Takođe, koristimo klase `KeyFactory`, `PublicKey` i `PrivateKey` za izgradnju ispravne specifikacije za zahtevani algoritam (*RSA*). Nakon, oporavljanja ključeva, ključevi će moći ponovo da se koriste za različite operacije sa asimetričnim algoritmima.

```
public class UcitajJavniPrivatniKljuc {
    public static void main(String[] args) throws IOException, NoSuchAlgorithmEx
        //Učitavanje javnog i privatnog ključa
        byte javni[] = ucitajKljuc("JavniKljuc.JK");
        byte privatni[] = ucitajKljuc("PrivatniKljuc.PK");

        //Oporavljanje javnog ključa
        X509EncodedKeySpec kljucSpecJ = new X509EncodedKeySpec(javni);
        System.out.println("Javni ključ je uspešo učitan!");
        System.out.println("Format javnog ključa: "+kljucSpecJ.getFormat());
        KeyFactory fabrika = KeyFactory.getInstance("RSA");
        PublicKey javniKljuc = fabrika.generatePublic(kljucSpecJ);
        System.out.println("Javni ključ: \n"
                +new BigInteger(javniKljuc.getEncoded()));

        //Oporavljanje privatnog ključa
        PKCS8EncodedKeySpec kljucSpecP = new PKCS8EncodedKeySpec(privatni);
        System.out.println("Privatni ključ je uspešo učitan!");
        System.out.println("Format privatnog ključa: "+kljucSpecP.getFormat());
        PrivateKey privatniKljuc = fabrika.generatePrivate(kljucSpecP);
        System.out.println("Privatni ključ: \n"
                +new BigInteger(privatniKljuc.getEncoded()));

    }
    public static byte [] ucitajKljuc(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
}
```

*Primer 3.4 Učitavanje javnog i privatnog ključa iz datoteke*

Programski izlaz:

```
Javni ključ je uspešo učitan!
Format javnog ključa: X.509
Javni ključ:
199582142956008510570996053172578267554407915993617735369142167915397142162101
938993683512527136149765777618971332379601017606608519615986651849761910039110
181671477891975286202064546690313191578807323523850017951400797019178933676779
182736289034985331168956434809238446734415440638353791329153627254751851939369
608045621860468795703297
Privatni ključ je uspešo učitan!
Format privatnog ključa: PKCS#8
Privatni ključ:
127529353104145855604848801036866616280166079002254795915728875880062449615574
616966763311037845999463201443812406418100243830836797104435354870144362004787
619208032929238722707681257221512858888916253823066036518442636507250273859748
537938263757161275348452459888872534553270094848277996183148427430879006737959
531346729452750566220492963606396070285512167372190831367375185595568414959493
177487913938469499310196302217315199394508255854614048697919037458492508417379
380882540721883683938716622302892427578135037633847210692227691931341646977901
073072170425754399379588831606827790122797757693869543947834852513130363829495
306843595338880001499715020688228635487012433309019632005517883285044121932361
994645472917192132347482877566950227512460133079450105639997036851997688123310
558198478020839371906778220612354200998664904172206023312543360360071051666873
312838250433990291787722813929585936021237669701642758939503520928832989896955
852488126457552795525729800336145724265449753695280459111255325213530155950976
755309524332324955610286716688174745479071524343452316231156591765436446293156
696600482850101226826583812637891854992503431847345666445073010545124817058780
623748270931267620537771078801712942376454038617077395664294421553984348347374
452929925937936163967373794612819185674031830309213303264168141318263975711990
252963071655725932971983
```

Sada, kada smo razumeli proces generisanja ključeva asimetričnih ključeva, proces zapisivanja u datoteku i ponovnog učitavanja iz datoteke u instancu ključa, prelazimo na klasu `Cipher` i funkcionalnosti podržane za operacije sa asimetričnim šiframa.

## 3.2 ASIMETRIČNE ŠIFRE – ŠIFARSKA KLASA

Slično kao i kod simetričnih šifri, `Cipher` klasa se koristi za inicijalizovanje šifarskog algoritma, pritom oslanjajući se na par generisanih ključeva kod asimetričnih šifri. Obzirom da sve asimetrične šifre ne podržavaju šifrovanje, u ovom delu baziraćemo se na RSA algoritmu. Navešćemo osnovne algoritamske specifikacije koje se odnose na asimetrične šifre sa `Cipher` klasom:

- RSA/ECB/PKCS1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

Sledi jedna linija koda za inicijalizovanje šifarske instance:

```
Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding") ;
```

Preko algoritamske specifikacije *RSA/ECB/PKCS1Padding*, koristimo *ECB* režim za šifrovanje i *PKCS1Padding* standard za dopunjavanje otvorenog teksta. Zatim, preko metoda `init()`, slično kao i kod simetričnih šifri inicijalizujemo operaciju šifrovanje ili dešifrovanje. Sledi fragment koda, gde u prvoj liniji inicijalizujemo operaciju šifrovanja i javni ključ, a u drugoj liniji šifrujemo otvoreni tekst:

```
Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding");
sifra.init (Cipher.ENCRYPT_MODE, javniKljuc);
```

Na izgled, poziv metoda izgleda identično kao kod simetričnih šifri, međutim kod asimetričnih šifri moramo da pazimo na ključ koji smo prosledili (javni ili privatni). Takođe, ako želimo programski da dobijemo sve informacije o podržanim algoritmima u standardnom *JCE* provajderu, to možemo da uradimo izvršavanjem sledećeg fragmenta Java koda:

```
public class SifarskaKlasa {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoS
        Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        System.out.println("Informacije: "+sifra.getProvider().getInfo());
    }
}
```

*Primer 3.5 Pribavljanje informacija o provajderu*

Programski izlaz:

```
Informacije: SunJCE Provider (implements RSA, DES, Triple DES, AES,
Blowfish, ARCFOUR, RC2, PBE, Diffie-Hellman, HMAC)
```

U primeru koji sledi, odlučili smo da koristimo operaciju šifrovanje sa *RSA* šifrom, u tom slučaju koristićemo javni ključ druge strane.

### 3.2.1 ŠIFROVANJE SA JAVNIM KLJUČEM - *RSA*

Kao što smo već naveli, ukoliko želimo da pošaljemo šifrovanu poruku sa *RSA* šifrom, potrebno je da posedujemo kod sebe javni ključ druge strane kojoj želimo poslati šifrovanu poruku. U primeru koji sledi, šifrovaćemo datoteku „poruka.txt“:

```
public class RSAsifrovanje {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //Učitavanje ključa iz datoteke
        byte javni[] = ucitajKljuc("JavniKljuc.JK");
        X509EncodedKeySpec kljucSpecJ = new X509EncodedKeySpec(javni);
        System.out.println("Javni ključ je uspešo učitan!");
        System.out.println("Format javnog ključa: "+kljucSpecJ.getFormat());
        KeyFactory fabrika = KeyFactory.getInstance("RSA");
        PublicKey javniKljuc = fabrika.generatePublic(kljucSpecJ);
        //Inicijalizovanje šifre
        Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        sifra.init(Cipher.ENCRYPT_MODE, javniKljuc);
        //Učitavanje poruke iz datoteke
        byte poruka[] = ucitajPoruku("poruka.txt");
        System.out.println("Poruka: "+new String(poruka));
        //Šifrovanje poruke
        byte sifrat[] = sifra.doFinal(poruka);
        System.out.println("Sifrat: "+new String(sifrat));
    }
    public static byte [] ucitajKljuc(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
    public static byte [] ucitajPoruku(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
}
```

*Primer 3.6* RSA *šifrovanje*

Programski izlaz:

```
Javni ključ je uspešo učitan!
Format javnog ključa: X.509
Poruka: Univerzitet Singidunum
□ʌ□-□ &f□h□□ □□□5□ □ 7□5K□□SfM□□□8#)□□T]□F□□.□□j!* tf□C]%□SF□ 1□□ □
, □ h□□j"□□□□ 'Tデ□]&□ ȝj□□□ Z $F!(□] l□\a□v □/□K□5TKD□□G
```

## 3.2.2 DEŠIFROVANJE SA PRIVATNIM KLJUČEM -RSA

U sledećem primeru, simuliraćemo kriptografski proces na drugoj strani komunikacionog kanala, u kojoj je zadatak dešifrovanje sa privatnim ključem preko *RSA* algoritma. Slično kao i u prethodnom primeru, učitaćemo privatni ključ i dešifrovati šifrat.

```
public class RSAdesifrovanje {
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //Učitavanje privatnog ključa
        byte privatni[] = ucitajKljuc("PrivatniKljuc.PK");
        PKCS8EncodedKeySpec kljucSpecP = new PKCS8EncodedKeySpec(privatni);
        KeyFactory fabrika = KeyFactory.getInstance("RSA");
        System.out.println("Privatni ključ je uspešo učitan!");
        System.out.println("Format privatnog ključa: "+kljucSpecP.getFormat());
        PrivateKey privatniKljuc = fabrika.generatePrivate(kljucSpecP);
        //Inicijalizovanje šifre
        Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        sifra.init(Cipher.DECRYPT_MODE, privatniKljuc);
        //Učitavanje šifrata iz datoteke
        byte sifrat[] = ucitajSifrat("sifrat.txt");
        //Dešifrovanje poruke
        byte poruka[] = sifra.doFinal(sifrat);
        System.out.println("Poruka: "+new String(sifrat));
    }
    public static byte [] ucitajKljuc(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
    public static byte [] ucitajSifrat(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
}
```

Programski izlaz:

```
Privatni ključ je uspešo učitan!
Format privatnog ključa: PKCS#8
Poruka: Univerzitet Singidunum
```

U nastavku ovog dela pokazaćemo preko kojih metoda jednostavno možemo da uporedimo jednakost ključeva. Provajderi će obično ponuditi dobro osmišljeno rešenje koje pravilno implementira metode `equals()`, `hashcode()` i `clone()`, kao i implementacija Java `Serializable` interfejsa (interfejs se koristi za serijalizaciju stanja objekta).

U slučaju da je potrebno utvrditi jednakost ključeva preko metoda `equals()` (metod nije implementiran od strane provajdera), najbolji način u praksi da se odredi jednakost ključeva je provera svakog bajta (poređenje bajta sa bajtom). Da ne bismo koristili petlje, koristićemo u sledećem primeru `java.util.Arrays` pomoćnu klasu.

```
equals = Arrays.equals(javniKljucA.getEncoded(), privatniKljuc.getEncoded());
```

Postoji slučaj u kome je potrebno uraditi proveru jednakosti između ključeva različitog formata. To je slučaj kada je jedan ključ instanca `RSAPrivateKey`, a drugi `RSAPrivateCrtKey` klase. U tom slučaju poređenje po jednakosti se vrši preko pojedinačnih podataka ili meta-podataka sa metodom `equals()`.

## 3.3 PROTOKOLI ZA RAZMENU TAJNOG KLJUČA

Sledeći problem je razmena ključa u simetričnim šifarskim sistemima i pronalaženje sigurnog načina za razmenu simetričnih ključeva između dve udaljene lokacije. Na primer, banka treba da obavi zaštićenu transakciju sa klijentom, ali kako dostaviti ključ? Najbezbednije je da se dostava ključa obavi lično. Kod ovog načina dostavljanja veliki problem predstavlja potrebno vreme za dostavu ključa. Manje bezbedno je angažovanje kurirskih službi u te svrhe. Postavlja se pitanje da li je to nezavisna organizacija i da li je to slaba karika u sistemu?

Sličan scenario jeste dostava ključa vojnim jedinicama u ratnim uslovima, dostava ključa nuklearnim podmornicama koje se nalaze na 1000 kilometara od vojne baze. Međutim, država raspolaže novcem, resursima i može da se izbori sa ovakvim problemima, ali za civilni sektor je ovo bio gotovo nerešiv problem.

Razmotrićemo sada jedan klasičan problem. Alisa i Bob žele da imaju sigurnu komunikaciju. Zna se ili pretpostavlja da Trudi prisluškuje njihov komunikacioni kanal. Alisa i Bob mogu povremeno da se lično sastaju i razmenjuju ključeve za naredni period komuniciranja. Ovo rešenje je uglavnom teško izvodljivo u praksi, a može da bude i nemoguće.

Difi i Helman su prvi predložili primenu asimetričnog šifarskog sistema. Za šifrovanje i dešifrovanje koristio bi se različit ključ.

### 3.3.1 DIFI-HELMAN PROTOKOL

Difi-Helman protokol predložili su Vitfild Difi (*Whitfield Diffie*) i Martin Helman (*Martin Hellman*). Skraćenicu „*DH*“ uglavnom ćemo koristiti za Difi-Helman algoritam ili protokol za razmenu ključa. Protokol je razvijen nezavisno na dva mesta. Predstavlja algoritam za razmenu zajedničkih simetričnih ključeva. Nije namenjen za šifrovanje ili digitalno potpisivanje. Sigurnost *DH* algoritma zasniva se na računski složenom računanju (jednosmerne funkcije) diskretnog algoritma za zadate $g, p$ i $g^n mod\ p$ naći $n$. Za poznato $g$ i $x$, gde je $x = g^n$, može da se odredi $n$:

$$n = \log_g(x).$$

Ako je $x = g^n (\mathrm{mod}\ p)$, $n$ se takođe određuje preko diskretnog logaritma. U nastavku ćemo proći kroz matematičke osnove *DH* algoritma. Neka je $p$ veliki prost broj i $g$ takvo da se za svako $x \in \{1,2, \dots, p-1\}$ može naći $n$ tako da je:

$$x = g^n mod\ p$$

Vrednosti $p$ i $g$ su javne. U komunikaciji između Alise i Boba, ove parametre može da odredi bilo ko od njih, a zatim se parametri razmenjuju javnim kanalom. Alisa bira

svoju tajnu vrednost $a$. Bob bira svoju tajnu vrednost $b$. Ove vrednosti predstavljaju velike slučajne proste brojeve.

U prvoj rundi, Alisa javno šalje vrednost $g^a \pmod p$ Bobu. Bob javno šalje vrednost $g^b \pmod p$ Alisi. U drugoj rundi, oboje na osnovu primljenih vrednost računaju zajedničku tajnu vrednost $g^{ab} \pmod p$. Dobijena zajednička tajna vrednost može da se koristi kao simetrični ključ *K*. Pretpostavimo da Alisa i Bob koriste vrednost $g^{ab} \pmod p$ kao tajni ključ *K*. Trudi može da sazna vrednosti $g^a \pmod p$ i $g^b \pmod p$ jer su poslate javnim kanalom. Međutim, Trudi ne može na osnovu ovih vrednosti da sazna vrednost $a$ ili $b$, a ako bi to mogla da uradi, sistem bi bio razbijen, tada bi to značilo da je rešila problem diskretnog logaritma.

Standardom *PKSC#3*, definisane su i preporučene vrednosti parametara $a, b, g$ i $p$ za generisanje i razmenu tajnih ključeva. U praksi se za $p$ vrednost koristi veliki broj koji je veći od 1024 bita. Napomenućemo da su sve metode definisane u klasi `KeyAgreement` u paketu `javax.crypto`.

Sledi primer koda za generisanje parametara $p$ i $g$ za DH protocol:

```
public class DHparametri {
    public static void main(String[] argv) throws Exception {
    AlgorithmParameterGenerator paramGen =
            AlgorithmParameterGenerator.getInstance("DH");
    paramGen.init(1024);

    AlgorithmParameters params = paramGen.generateParameters();
    DHParameterSpec dhSpec =
            (DHParameterSpec) params.getParameterSpec(DHParameterSpec.class);

    System.out.println("Javni parametar P: \n" + dhSpec.getP());
    System.out.println("Javni parametar G: \n" + dhSpec.getG());
    System.out.println("Veličina eksponenta L: \n" + dhSpec.getL());
    }
}
```

*Primer 3.7 Generisanje DH parametara*

Kao što vidimo u primeru (Primer 3.7), parametri DH protokola generisani su vrlo jednostavno preko klase `DHParameterSpec`.

Programski izlaz:

```
Javni parametar P:
179444214689727505035817344799946747356192811391721103090490545365751447636308338395858766144330568452840319047175425752064227652967954605575810261198499732034603993693549758004520683774083369360421971999262652020197892666708683925514685582745391483520518936163035944287884683045305514466602586718369600839560900
Javni parametar G:
465768740384293101279468380491775030454314008555158616438509888062226451890083414586504543839957473658612858587222252543417369560832947485215026969963122206317015961143933059716954432936478796160980606590750409731536828765192833942391864014831750724599878004869713597292837627406333040876709731726861102707040
Veličina privatnog eksponenta L:
1023
```

Uprkos svom jednostavnom interfejsu, protokol za razmenu simetričnog ključa može biti veoma složen. *SunJCE* provajder implementira Difi-Helman algoritam za razmenu ključa (engl. *Key Agreement*). Ovaj algoritam je zasnovan na sledećem protokolu koji se sastoji iz osam koraka:

1. Alisa (kao prva u protokolu) generiše DH parametre, javni i privatni ključ;
2. Alisa šalje Bobu javni ključ i parametre *DH* algoritma;
3. Bob na osnovu *DH* parametara generiše svoj javni ključ i privatni ključ, zatim šalje Alisi svoj javni ključ;
4. Alisa sa svojim privatnim ključem i Bobovim javnim ključem generiše tajni ključ *K* (`SecretKey`). Ovde su zahtevane dve faze: jedna u kojoj koristi svoj privatni ključ i druga u kojoj šalje Bobu svoj javni ključ;
5. Bob obavlja identične operacije sa svojim privatnim ključem i Alisinim javnim ključem. U ovoj fazi protokola obe strane poseduju zajednički tajni ključ;
6. Bob i Alisa konvertuju tajni ključ u *DES* ključ;
7. Alisa koristi tajni ključ za šifrovanje poruke koju šalje Bobu;
8. Bob koristi tajni ključ za dešifrovanje poruke;

Poslednje dve faze predstavljaju upotrebu simetrične kriptografije, u kojima obe strane mogu poslati šifrovanu poruku i primiti dešifrovanu poruku.

U primeru (Primer 3.8), simuliraćemo preko programskih niti scenario za razmenu ključeva preko DH protokola. Sledi primer:

```
public class DHProtokol implements Runnable {
    byte bob[], alisa[];
    boolean krajAlisa = false;
    byte[] sifrat;
    BigInteger alisaP, alisaG;
    int alisaL;
    public static void main(String args[]) {
        DHProtokol test = new DHProtokol();  //Prva počinje Alisa
        new Thread(test).start();            //Počinje Bob
        new Thread(test).start();
    }
    public synchronized void run() {
        if (!krajAlisa) {
            krajAlisa = true;
            Alisa();
        } else {
            Bob();
        }
    }
    public synchronized void Alisa() {...40 lines }

    public synchronized void Bob() {...41 lines }
}
```

*Primer 3.8 Simulacija* DH *protokola preko programskih niti*

Ukoliko posmatramo ovaj protokol sa bezbednosnog aspekta, zaključićemo da protokol nije bezbedan, jer bilo ko može da se predstavi kao Alisa ili kao Bob. Da bi smo sprečili ovu vrstu napada (Čovek u sredini), potrebno je da dodatno potpišemo javne ključeve koji se šalju preko javnog kanala. U tom slučaju sprečili smo napad „Čovek u sredini“. Svakako, javni ključ je dovoljno samo potpisati, ali ne i šifrovati. Bezbednost DH protokola nije zasnovana na tajnosti parametara koji se razmenjuju između dve strane.

```
public synchronized void Alisa() {
    try {
        //1 korak:  Alisa generiše par ključeva (javni i privatni)
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
        kpg.initialize(1024);
        KeyPair kp = kpg.generateKeyPair();
        //2 korak:  Alisa šalje svoj javni ključ i DH parametre Bobu
        Class dhClass = Class.forName("javax.crypto.spec.DHParameterSpec");
        DHParameterSpec dhSpec = ((DHPublicKey) kp.getPublic()).getParams();
        alisaG = dhSpec.getG();
        alisaP = dhSpec.getP();
        alisaL = dhSpec.getL();
        alisa = kp.getPublic().getEncoded();
        notify();
        //4 korak,1 deo:Alisa obavlja prvu fazu sa svojim privatnim ključem
        KeyAgreement ka = KeyAgreement.getInstance("DH");
        ka.init(kp.getPrivate());
        //4 korak,2 deo:Alisa obavlja drugu fazu sa Bobovim javnim ključem
        while (bob == null) {
            wait();
        }
        KeyFactory kf = KeyFactory.getInstance("DH");
        X509EncodedKeySpec x509Spec = new X509EncodedKeySpec(bob);
        PublicKey pk = kf.generatePublic(x509Spec);
        ka.doPhase(pk, true);
        //4 korak,3 deo:Alisa izračunava tajni ključ
        byte secret[] = ka.generateSecret();
        //6 korak:Alisa generiše DES ključ
        SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
        DESKeySpec desSpec = new DESKeySpec(secret);
        SecretKey key = skf.generateSecret(desSpec);
        //7 korak:Alisa šifruje i šalje šifrovane podatke Bobu
        Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
        c.init(Cipher.ENCRYPT_MODE, key);
        String poruka = "Zdravo Bobe, UNIVERZITET SINGIDUNUM!";
        sifrat = c.doFinal(poruka.getBytes());
        notify();
    } catch (NoSuchAlgorithmException | ClassNotFoundException | InvalidKeyE
    }
}
```

*Primer 3.9 Programski kod na Alisinoj strani* DH *protokola*

U primeru (Primer 3.9), Alisa generiše sve neophodne parametre za DH protokol sa Bobom. Nakon uspostave tajnog ključa preko DH protokola, Alisa šifruje poruku „Zdravo Bobe, UNIVERZITET SINGIDUNUM“ sa DES algoritmom. Zatim, šalje šifrat Bobu. Na Bobovoj strani se odvija inverzan proces.

```
public synchronized void Bob() {
    try {
        //3 korak:Bob koristi parametre koje je dobio od Alise
        //za aračunanje javnog i privatnog ključa
        while (alisa == null) {
            wait();
        }
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
        DHParameterSpec dhSpec = new DHParameterSpec(alisaP,alisaG,alisaL);
        kpg.initialize(dhSpec);
        KeyPair kp = kpg.generateKeyPair();
        bob = kp.getPublic().getEncoded();
        notify();
        //5 korak,1 deo:Bob koristi svoj privatni ključ za obavljanje
        //prve faze protokola
        KeyAgreement ka = KeyAgreement.getInstance("DH");
        ka.init(kp.getPrivate());
        //5 korak,2 deo:Bob koristi Alisin javni ključ za obavljanje
        //druge faze protokola
        KeyFactory kf = KeyFactory.getInstance("DH");
        X509EncodedKeySpec x509Spec = new X509EncodedKeySpec(alisa);
        PublicKey pk = kf.generatePublic(x509Spec);
        ka.doPhase(pk, true);
        //5 korak,3 deo:Bob izračunava tajni ključ
        byte secret[] = ka.generateSecret();
        //6 korak:Bob generiše DES ključ
        SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
        DESKeySpec desSpec = new DESKeySpec(secret);
        SecretKey key = skf.generateSecret(desSpec);
        //8 korak:Bob prima šifrovani tekst i dešifruje ga
        Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
        c.init(Cipher.DECRYPT_MODE, key);
        while (sifrat == null) {
            wait();
        }
        byte plaintext[] = c.doFinal(sifrat);
        System.out.println("Bob dešifruje poruku:" + new String(plaintext));
    } catch (IllegalStateException | InterruptedException | InvalidAlgorithm

    }
}
```

*Primer 3.10 Programski kod na Bobovoj strani* DH *protokola*

Programski izlaz:

```
Bob dešifruje poruku:Zdravo Bobe, UNIVERZITET SINGIDUNUM!
```

# 4. HEŠ FUNKCIJE, INTEGRITET PORUKE I DIGITALNI POTPIS

Heš funkcije (engl. *Hash*) se ne koriste za šifrovanje. Za razliku od simetrične i asimetrične šifre, operacija hešovanja ne zahteva ključ. Pogodna svojstva ovih funkcija koriste se za uspešnu realizaciju kriptografskih operacija koje se odnose na hešovanje poruke (engl. *Message Digest*), integritet prenesenih poruka (engl. *Message Authentication Code*) i digitalni potpis (engl. *Digital Signature*).

Njihova jednosmernost, predstavlja prednost u odnosu na druge funkcije, koja nastaje kao posledica kompresije sa gubicima. Kod operacije šifrovanja, najčešće veličina šifrata odgovara veličini otvorene poruke (simetrične blok šifre), dok su dobijene heš vrednosti preko jednosmernih funkcija dužinski konstantne, bez obzira na ulaznu količinu podataka. Takođe, u kriptografskim funkcijama za šifrovanje, postoji funkcija za dešifrovanje (inverzna funkcija funkciji šifrovanja) ukoliko se poseduje odgovarajući ključ, dok kod heš funkcija ne postoji inverzna funkcija. Ovo je razlog zašto ove funkcije oslovljavamo sa jednosmerne funkcije.

Pošto je ulaz proizvoljne dužine koja znatno nadmašuje izlaz (heš vrednost), verovatno je da postoji veliki broj kolizija. Na primer, pretpostavimo da heš funkcija generiše izlaz od 128 bitova. Pretpostavimo da je za ulaz u funkciju uzeta poruka dužine 160 bitova, a to je 32 bita duže od izlaza. Sledi da postoji još $2^{32} \approx 4.000.000.000$ mogućih ponavljanja. Otpornost na kolizije podrazumeva da je računarski veoma kompleksno naći jednu od njih.

Najznačajnija primena heš funkcije je kod kriptografskih operacija za digitalni potpis. Na primer, Alisa potpisuje otvorenu poruku $M$ šifrovanjem na bazi privatnog ključa , tj. $S = [M]_{\text{Alisa}}$. Dalje, Alisa šalje otvorenu poruku $M$ i potpis $S$, a Bob verifikuje digitalni potpis šifrovanjem dobijenog potpisa Alisinim javnim ključem, tj. $M = \{S\}_{\text{Alisa}}$. Ukoliko je poruka $M$ velike dužine javljaju se dva problema. Prvi problem se odnosi na kompleksnost digitalnog potpisivanja i verifikacije potpisa, tj. duža poruka se duže obrađuje, a poznato je da su algoritmi za potpisivanje poprilično spori. Drugi problem se odnosi na dodatno opterećivanje komunikacionog kanala, tj. za određeni propusni opseg potrebno je preneti dva puta više bitova (i poruka i potpis).

## 4.1 KRIPTOGRAFSKE HEŠ FUNKCIJE

Heš funkcija u najosnovnijem obliku predstavlja sumu poruke. Na primer, to može biti suma heksadecimalnih vrednosti. Ako je data poruka {0x270 x280 x29}, tada je njena suma 84. Problem u ovom slučaju je što se lako može naći druga kombinacija sadržaja koja daje isti heš (na primer, poruka: 0x250 x280 x31). Drugim rečima, isti heš se može dobiti za više različitih poruka, što nije dobro.

Kriptografski cilj je da jedan heš odgovara samo jednoj poruci, a to znači da je jedinstven. Heš funkcije se standardno primenjuju kod provere integriteta primljene poruke.

Kriptografska heš funkcija $h(x)$ mora da poseduje sledeća svojstva:

- **Kompresija** – Za proizvoljan broj bitova ulazne poruke $x$, dužina dobijene heš vrednosti $y = f(x)$ je konačna i najčešće manja od same ulazne poruke (na primer 128, 160 bitova i sl.).

- **Efikasnost** - heš vrednost $h(x)$treba da se jednostavno izračuna za bilo koji ulaz $x$. Kompleksnost izračunavanja, naravno, zavisi od dužine ulazne poruke, ali ne značajno.

- **Jednosmernost** - heš funkcije su jednosmerne funkcije. To znači da je jednostavno izračunati heš vrednost za datu poruku, ali je veoma teško rekonstruisati poruku iz date heš vrednosti, tj. praktično je nemoguće.

- **Otpornost na kolizije** - kod dobre heš funkcije teško je naći sudar (koliziju). Ako je data poruka $x_1$ čija je heš vrednost $h(x_1)$ teško je naći drugu poruku $x_2$ tako da njihove sume budu jednake, tj. da je $h(x_1) = h(x_2)$.

- **Lavinski efekat** - promena samo jednog bita na ulazu treba da rezultuje u promenu bar polovine bita dobijene heš vrednosti.

### 4.1.1 *MD5* JEDNOSMERNA FUNKCIJA

MD5 (engl. *Message-Digest algorithm 5*) heš funkcija predstavlja primer prave kriptografske heš funkcije koja se najčešće koristila. Danas se zna da ova heš funkcija ima slabosti i da je neotporna na kriptografske napade, tako da se ređe primenjuje u kriptografiji. Još uvek ima svoju namenu u proveri integriteta većih fajlova (pre svega zbog svoje brzine rada). Dužina dobijene heš vrednosti (sažetka, sume) je 128 bitova. Tipično se predstavlja sa 32 heksadecimalne cifre.

*MD5* algoritam je razvio Ronald Rivest 1991. godine. Baziran je na prethodnom *MD4* algoritmu i nešto je sporiji od njega. Danas se zna da je podložan na *„brute force birthday attack“*. Jedan takav projekat pod imenom *MD5CRK* je pokrenut 1. marta 2004. godine sa namerom da dokaže slabost *MD5* algoritma. Nedugo zatim, 17. avgusta 2004. godine, objavljeno je da su Ksiaoun Vang, Denguo Feng, Ksuejia Lai i Ksongbo Ju uspešno razbili algoritam odnosno da su pronašli koliziju na algoritmu. Za razbijanje ovog algoritma bio im je potreban samo jedan sat na *IBM* p690 klasteru. 2005. godine Arjen Lenstra, Ksiaoun Vang, i Bene de Veger su demonstrirali su kreiranje dva *X.509* sertifikata sa različitim javnim ključevima i istom *MD5* heš vrednosti. Nekoliko dana potom Vlastimil Klima je kreirao unapređeni algoritam koji je u stanju da na običnom *PC* računaru za nekoliko sati kreira koliziju *MD5* algoritma. Danas je to moguće uraditi za jedan minut. U svakom slučaju, *MD5* funkcija je obeležila prošla vremena, kada je reč o heš vrednostima.

### 4.1.2 SHA JEDNOSMERNA FUNKCIJE

*SHA* (engl. *Secure Hash Algorithm*) je klasa kriptografskih funkcija za sažimanje. U ovoj grupi kriptografskih funkcija najčešće korišćena funkcija je *SHA*-1 koja je našla primenu u velikom broju kriptografskih protokola kao što su *TLS, SSL, PGP, SSH, S/MIME* i *IPSec*. *SHA-1* se može smatrati naslednikom *MD5* algoritma. *SHA* algoritmi su kreirani od strane američke agencije za bezbednost (*NSA*) i publikovani kao zvanični standard vlade *SAD*. Heš vrednost je dužine 160 bitova.

Prvi član grupe *SHA* algoritama, publikovan je 1993. godine, i zvanično nazvan *SHA*, ali se on često u literaturi naziva *SHA-0*, da bi se sprečila zabuna sa njegovim kasnijim

naslednicima. Dve godine kasnije nastaje *SHA*-1, prvi naslednik *SHA*. Još četiri varijante ovog algoritma su publikovane i to *SHA*-224, *SHA*-256, *SHA*-384 i *SHA*-512 i oni se jednim imenom nazivaju *SHA*-2, a razlikuje ih samo dužina sažetka koji proizilazi iz njih.

Algoritmi *SHA*-256, *SHA*-384 i *SHA*-512 objavljeni su 2002. godine dok je varijanta *SHA*-224 objavljena 2004. godine. Ove duže varijante su znatno sigurnije u odnosu na *SHA*-0 i *SHA*-1, prvenstveno zbog dužine sažetka i određenih promena u algoritmu.

Nove verzije *SHA* algoritma koriste druge vrednosti pomeranja kao i dodatne konstante, ali u suštini nema značajnih promena u odnosu na prethodnike. I pored veće sigurnosti oni se i danas manje koriste u odnosu na *SHA*-1 koji predstavlja industrijski standard u kriptografiji.

*SHA*-0 i *SHA*-1 algoritmi se danas ne mogu smatrati bezbednim zato što postoje kriptografski napadi koji su sposobni da pronađu koliziju za relativno kratko vreme. U tabeli koja sledi, dat je pregled svih verzija *SHA* funkcije:

*Tabela 4.1 Verzije* SHA *heš funkcije*

| **Algoritam** | **Veličina sažetka u bitovima** | **Inicijalna veličina u bitovima** | **Veličina bloka u bajtovima** |
|---|---|---|---|
| ***SHA*-0** | 160 | | |
| ***SHA*-1** | 160 | 160 | 64 |
| ***SHA*-224** | 224 | 256 | 64 |
| ***SHA*-256** | 256 | 256 | 64 |

| | | | |
|---|---|---|---|
| **SHA-384** | 384 | 512 | 128 |
| **SHA-512** | 512 | 512 | 128 |

## 4.2 HEŠ FUNKCIJE – IMPLEMENTACIJA

*JCA* ekstenzija podržava rad sa kriptografskim jednosmernim funkcijama – heš funkcijama, preko klase `MessagesDigest`. Svaka implementacija Java platforme zahteva podršku za sledeće heš funkcije: *MD5* i *SHA*, funkcije koje su ujedno i najbolje, dok drugi provajderi mogu da ponude i druge algoritme iste namene.

Slede nazivi algoritama jednosmernih funkcija koji su dostupni u *JCE*:

- *MD2*
- *MD5*
- *SHA*-1
- *SHA*-224
- *SHA*-256
- *SHA*-384
- *SHA*-512

Sledi jedan primer poziva *MD5* algoritma za računanje heš vrednosti veličine 128 bitova. Generisaćemo otisak poruke ili heš vrednost za poruku 1 i poruku 2. Očekivano je da heš vrednosti budu identične.

```
public class Hes1md5 {
    public static void main(String[] args) throws NoSuchAlgorithmException, Unsu
        //1 deo
        //Poruka
        String poruka1 = "Univerzitet Singidunum,"
                + "Fakultet za informatiku i računarstvo,"
                + "Praktikum-zaštita informacionih sistema.";
        String poruka2 = "Univerzitet Singidunum,"
                + "Fakultet za informatiku i računarstvo,"
                + "Praktikum-zaštita informacionih sistema. ";
        //Inicijalizovanje MD5 jednosmerne funkcije
        MessageDigest md5 = MessageDigest.getInstance("MD5") ;
        byte[] hesMD5 = md5.digest(poruka1.getBytes("UTF-8"));
        System.out.println("Heš vrednost: "+Hex.toHexString(hesMD5));
        System.out.println("Korišćena funkcija: "+md5.getAlgorithm());
        System.out.println("Veličina otiska: "+hesMD5.length);
        //2 deo
        md5.reset();
        hesMD5 = md5.digest(poruka2.getBytes("UTF-8"));
        System.out.println("Heš vrednost: "+Hex.toHexString(hesMD5));
        System.out.println("Korišćena funkcija: "+md5.getAlgorithm());
        System.out.println("Veličina otiska: "+hesMD5.length);
    }
}
```

*Primer 4.1 Implementacija* MD5 *funkcije*

Programski izlaz:

```
Heš vrednost: 6095065bcb54e93112a2a48028dc3ce8
Korišćena funkcija: MD5
Veličina otiska: 16
Heš vrednost: 8fe2724d3e88ce621551a36e2110a422
Korišćena funkcija: MD5
Veličina otiska: 16
```

U izlazu iz primera (Primer 4.1) jasno vidimo da su otisci poruka potpuno različiti. Razlog tome su naizgled identične poruke. Međutim, poruka 2 sadrži jedan dodatni razmak na kraju. Preko ovog primera smo predstavili zahtevane kriptografske osobine jednosmernih funkcija koje smo prethodno naveli.

Konceptualno klasa `MessageDigest` radi potpuno identično kao i klasa `Cipher`. Napominjemo da klasa `Cipher`, obezbeđuje metode za jednodelne ili višedelne operacije. U slučaju višedelnih operacija, pozivaćemo metod `update()` sve dok

postoje novi delovi podataka, a zatim na kraju obavićemo završnu operaciju preko metoda `doFinal()`. Rezultat ove metode je niz bajtova koji predstavljaju šifrat ili otvorenu poruku u zavisnosti od primenjene operacije (šifrovanje ili dešifrovanje). Posle svih izvedenih operacija pozvaćemo metod `reset()`, da bismo pripremili instancu za izvođenje operacija nad novim podacima.

`MessageDigest` klasa takođe podržava jednodelne ili višedelne operacije, koje je moguće pozvati preko metoda `update()`, a završnu operaciju obaviti preko metoda `digest()`. Alternativno je moguće direktno pozvati metodu `digest()` ukoliko su prisutni svi potrebni podaci (urađeno u primeru iznad). Rezultat ovog metoda je standardno niz bajtova koji je vrlo jednostavno moguće konvertovati u heksadecimalni niz preko klase `Hex`, uvedene preko provajdera *bouncycastle*. Takođe, posle svih izvedenih operacija pozvaćemo metod `reset()`, da bismo pripremili instancu za izvođenje operacija nad novim podacima.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

U primeru (Primer 4.2), merili smo performanse *MD5*, *SHA*-1 i *SHA*-256 algoritma. Performanse algoritama su prikazane u jedinici vremena.

```
public static void main(String[] args) throws IOException, NoSuchAlgorithmEx
    //Učitavanje video datoteke
    byte datoteka [] = ucitajDatoteku("video.avi");
    System.out.println("Veličina datoteke u bajtovima: "+datoteka.length);
    //MD5 performanse
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    long pocetnoVreme = System.currentTimeMillis();
    byte[] MD5 = md5.digest(datoteka);
    long zaustavnoVreme = System.currentTimeMillis();
    System.out.println("MD5: "+Hex.toHexString(MD5));

    System.out.println("Potrebno vreme za MD5: "
            +((double)(zaustavnoVreme-pocetnoVreme)/1000)+"s");
    //SHA-1 performanse
    MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
    pocetnoVreme = System.currentTimeMillis();
    byte[] SHA1 = sha1.digest(datoteka);
    zaustavnoVreme = System.currentTimeMillis();
    System.out.println("SHA-1: "+Hex.toHexString(SHA1));
    System.out.println("Potrebno vreme za SHA1-1: "
            +((double)(zaustavnoVreme-pocetnoVreme)/1000)+"s");
}
```

*Primer 4.2 Performanse jednosmernih funkcija*

Korisnici kriptografskih operacija često razgovaraju o ceni tih operacija. Svakako, svaka operacija ima svoju cenu koju možemo izraziti u vremenu koje je potrebno za njeno izvršavanje. Ukoliko je potrebno neke operacije vremenski sinhronizovati, tada možemo trajanje neke operacije jednostavno odrediti pozivom metoda `System.currentTimeMillis()`.

Programski izlaz:

```
Veličina datoteke u bajtovima: 95494419
MD5: c9f2ae7d29ae7b7bf946c9337db19d28
Potrebno vreme za MD5: 0.305s
SHA-1: 507f7c47a54b89f85d912e1ba3e60b3eadc43d90
Potrebno vreme za SHA1-1: 0.49s
```

Sledeći primer (Primer 4.3), demonstrira upotrebu različitih algoritama za računanje heš vrednosti. Na izlazu iz programa, prikazani su rezultati korišćenih funkcija i veličina fajla za koga su heš vrednosti generisane.

```
public class HesDatoteka {
    public static void main(String[] args) throws IOException, NoSuchAlgorithmE
        //Učitavanje video datoteke
        byte datoteka [] = ucitajDatoteku("video.avi");
        System.out.println("Veličina datoteke u bajtovima: "+datoteka.length);
        //MD5
        MessageDigest md5 = MessageDigest.getInstance("MD5");
        byte[] MD5 = md5.digest(datoteka);
        System.out.println("MD5: "+Hex.toHexString(MD5));
        //SHA-1
        MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
        byte[] SHA1 = sha1.digest(datoteka);
        System.out.println("SHA-1: "+Hex.toHexString(SHA1));
        //SHA-224
        MessageDigest sha224 = MessageDigest.getInstance("SHA-224");
        byte[] SHA224 = sha224.digest(datoteka);
        System.out.println("SHA-224: "+Hex.toHexString(SHA224));
        //SHA-256
        MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
        byte[] SHA256 = sha256.digest(datoteka);
        System.out.println("SHA-256: "+Hex.toHexString(SHA256));
        //SHA-384
        MessageDigest sha384 = MessageDigest.getInstance("SHA-384");
        byte[] SHA384 = sha384.digest(datoteka);
        System.out.println("SHA-384: "+Hex.toHexString(SHA384));
        //SHA-512
        MessageDigest sha512 = MessageDigest.getInstance("SHA-512");
        byte[] SHA512 = sha512.digest(datoteka);
        System.out.println("SHA-512: "+Hex.toHexString(SHA512));
    }
    public static byte [] ucitajDatoteku(String nazivFajla) throws IOException{
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
}
```

*Primer 4.3 Računanje heša (*MD5, SHA-*1, 224, 256, 384, 512) za video datoteku*

Programski izlaz:

```
Veličina datoteke u bajtovima: 95494419
MD5: c9f2ae7d29ae7b7bf946c9337db19d28
SHA-1: 507f7c47a54b89f85d912e1ba3e60b3eadc43d90
SHA-224: 28a7bedd84c660cebd4b99857bdba0709245f27fea07876e3dff10cb
SHA-256: 3bb97b74116410d1c0d5bb20fef807f22efd6f89401d0be8a04cf974061db62e
SHA-384: ea215bea68bd7fbed7403beecc4cff31f6c5a441f1004f147b5347db54f82aace79abd2939787da4ee4
d4630ed04585c
SHA-512: e559b82ac1adfab5fb6a8530f1347d8721d7e25bd7cb41fd7125e0a73683da3de41a4a00687ed0f70f8
fa5f79e4db45f2dad13aad418abff84af0c7b1ba955b6
```

## 4.3 INTEGRITET PORUKE – *MAC*

Za obezbeđivanje funkcije integriteta poruke koja se prenosi primenjuje se kod za autentifikaciju poruke – *MAC* (engl. *Message Authentication Code*). U poglavlju o blokovskim šiframa razmatrana je jedna tehnika za generisanje *MAC*-a gde se blok šifra koristi u CBC režimu rada. Kako heš funkcija predstavlja sažetak poruke (engl. *message digest*) moguće je da se ova vrednost iskoristi za verifikaciju integriteta poruke. Ovakav pristup se označava sa *HMAC* (engl. *Hashed MAC*).

Za obezbeđivanje funkcionalnosti integriteta poruke $M$, Alisa izračunava heš vrednost $h(M)$ i šalje ka Bobu i $M$ i $h(M)$. Ukoliko se na prenosnom putu desi promena poruke iz $M$ u $M'$ Bob će jednostavno konstatovati da se to ne slaže sa dobijenom heš vrednosti $h(M)$. Međutim, razmatrani koncept nije dovoljan. Napadač Trudi može da zameni originalnu poruku $M$ sa svojom porukom $M'$, a da zatim izračuna i da zameni $h(M)$ sa $h(M')$. Na ovaj način Bob ne bi mogao da detektuje promenu podataka. Da bi se ovo izbeglo neophodno je da se heš vrednost šifruje. Moguće je da se koristi simetričan šifarski sistem sa tajnim deljenim ključem između Alise i Boba, $E(h(M), K)$. Drugi način je da se koristi asimetričan šifarski sistem, tj. u ovom slučaju $[h(M)]_{\text{Alisa}}$ .

Umesto šifrovanja koje je standardno kompleksan proces za realizaciju, moguće je koristiti *HMAC* bez šifrovanja. Kod ovog pristupa tajni deljeni ključ *K* se direktno dodaje poruci *M* pre izračunavanja heš vrednosti. Moguća su dva pristupa. Ključ se može dodati na početak ili na kraj poruke.

Snaga *HMAC* vrednosti zavisi od veličine tajnog ključa koji je korišćen za računanje vrednost. Najpoznatiji napad je napad sa potpunom pretragom tajnog ključa ili isprobavanjem svih mogućnosti. *HMAC* je znatno manje izložen napadu „sudaranja“

za razliku od osnovnih heš funkcija. *HMAC-MD5* ne pati od istih nedostataka koji su pronađeni u *MD5*.

### 4.3.1 GENERISANJE *HMAC* VREDNOSTI

Pre nego što izračunamo heš vrednost, potrebno je generisati tajni ključ koji će se koristiti. Standardno, koristićemo `SecretKey` i `KeyGenerator` klasu. Kada smo generisali ključ, tada možemo da inicijalizujemo *HMAC* algoritam preko klase `Mac`. Upotreba `Mac` klase je identična klasi `Cipher`, podržava režime za jednodelne ili višedelne operacije. U zavisnosti od izabranog režima, pozivamo metod `update()` više puta, a zatim završni metod `doFinal()` ili odmah ukoliko posedujemo sve podatke pozivamo `doFinal()` metod.

*HMAC* je algoritam razvijen na osnovi jednosmernih heš funkcija i iz tog razloga može se koristiti sa bilo kom heš funkcijom. Kompletna specifikacija *HMAC*-a nalazi se u *RFC* 2104 dokumentu. Svaka implementacija podržava sledeće *MAC* algoritme:

- *HmacMD5*
- *HmacSHA1*
- *HmacSHA256*

U primeru (Primer 4.4), pokazaćemo jednu implementaciju *HMAC* algoritma, koji implementira module za generisanje ključa i računanje *HMAC* heša. Korišćene su klase `KeyGenerator`, `SecretKey` i klasa `Mac` za poruku „Univerzitet Singidunum“.

```
public class HMAC {
    public static void main(String[] args) throws NoSuchAlgorithmException, Inva
        //Generisanje ključa
        KeyGenerator kg = KeyGenerator.getInstance("HmacSHA1");
        SecretKey kljuc = kg.generateKey();
        //Inicijalizovanje HMAC algoritma
        Mac mac = Mac.getInstance("HmacSHA1");
        mac.init(kljuc);
        String poruka = "Univerzitet Singidunum";
        byte hmac [] = mac.doFinal(poruka.getBytes("UTF-8"));
        System.out.println("HMAC algoritam: "+mac.getAlgorithm());
        System.out.println(Hex.toHexString(hmac));
        System.out.println("Veličina heša (bit): "+(hmac.length*8));
    }
}
```

*Primer 4.4 Računanje* HMAC-*a* SHA-*1*

Programski izlaz:

```
HMAC algoritam: HmacSHA1
9b190109a8a96dd8150ab727a0aaf6629b67ede2
Veličina heša (bit): 160
```

Iz ovog primera preko programskog izlaza, vidimo da je veličina heš vrednosti 160 bitova. U scenariju razmene šifrovanih poruka između Alise i Boba, Bob kada dobije šifrat i heš vrednost, uz pomoć svog tajnog ključa izračunaće novi *HMAC* heš i uporediti dobijeni šifrat. Ukoliko se dogodi napad „Čovek u sredini“, Bob će vrlo lako detektovati.

Drugi popularan način za generisanje tajnog ključa je preko *CSPRNG*, zaobliaženjem `KeyGenerator` klase. U tom slučaju, java implementacija je prikazana u sledećem primeru (Primer 4.5). Korišćene su klase `Securerandom`, `SecretKeySpec` i klasa `Mac` za poruku „Univerzitet Singidunum“.

```
public class HMAC_CSPRNG {
     public static void main(String[] args) throws NoSuchAlgorithmException, In
        //Generisanje ključa
        SecureRandom csprng = SecureRandom.getInstance("SHA1PRNG");
        byte[] materijalKljuca = new byte[16];
        csprng.nextBytes(materijalKljuca);
        SecretKeySpec kljuc = new SecretKeySpec(materijalKljuca, "HmacSHA1") ;
        //Inicijalizovanje HMAC algoritma
        Mac mac = Mac.getInstance("HmacSHA1");
        mac.init(kljuc);
        String poruka = "Univerzitet Singidunum";
        byte hmac [] = mac.doFinal(poruka.getBytes("UTF-8"));
        System.out.println("HMAC algoritam: "+mac.getAlgorithm());
        System.out.println(Hex.toHexString(hmac));
        System.out.println("Veličina heša (bit): "+(hmac.length*8));
    }
}
```

*Primer 4.5 Generisanje* HMAC *ključa preko* CSPRNG *izvora slučajnosti*

Programski izlaz:

```
HMAC algoritam: HmacSHA1
64c056caec9139b69d2184cdb9d9b99df83619d6
Veličina heša (bit): 160
```

Na osnovu ovih primera za *HMAC*, jasno je da se tajni ključ ne koristi za šifrovanje, kao kod simetričnih šifri (AES, DES). Međutim, bezbedna razmena ključa je neophodna u oba slučaja. Verovatno je najbolje rešenje uspostava protokola za razmenu simetričnih ključeva (*KeyAgreement*). U ovo svrhu moguće je koristiti DH protokol.

Bez obzira što smo pravilno implementirali *HMAC* algoritma za proveru integriteta poruke, takođe postoji potreba za uvođenjem vremenskih oznaka u cilju sprečavanja napada ponovnim slanjem poruka. U ovom slučaju, Bob će na osnovu vremenske oznake zadržati ili odbaciti poruku. Čak, iako je lažirana vremenska oznaka, preko *HMAC* heš vrednosti Bob će utvrditi ponovo da je poruka za odbacivanje.

## 4.3 DIGITALNI POTPIS

Digitalni potpis predstavlja jednu od najkorisnijih funkcionalnosti javne kriptografije. Digitalno potpisivanje elektronskih dokumenata odgovara svojeručnom potpisu na papiru, ali ima i niz drugih prednosti. Digitalni potpis je sastavni deo dokumenta, dok se potpis na papiru može dobiti njegovim kopiranjem sa drugog dokumenta. Verifikaciju potpisa na papiru može da potvrdi samo obučeni stručnjak, dok je verifikacija digitalnog potpisa jednoznačna i lako izvodljiva za svakog ko ima pristup Bobovom javnom ključu.

U *PKI* sistemima tradicija je da se otvorena poruka koja se šifruje obeležava sa *M*. Kada se koriste likovi iz kriptologije, tada Alisa i Bob treba da imaju svoj par ključeva (javni i privatni).

Koristićemo sledeću notaciju: javni ključ je $(N, e)$, a privatni je $d$. Ukoliko poruku označimo sa *M*, digitalni potpis sa *S*, onda je:

$$S = M^d \pmod N$$

Kod *RSA* algoritma, dešifrovanje i potpisivanje su iste operacije. Za računanje digitalnog potpisa *S* neophodno je poznavanje privatnog ključa *d*. Potvrda ispravnosti ili verifikacija digitalnog potpisa na poruci *M* može da se uradi na sledeći način:

$$S^e \pmod N = (M^d)^e \pmod N = M$$

Možemo primetiti da je operacija ista kao i kod šifrovanja. Svako ko zna $(N, e)$ može da potvrdi ispravnost digitalnog potpisa.

Na primer, Alisa želi da pošalje digitalno potpisanu poruku *M* Bobu. U ovom momentu Alisi i Bobu nije bitna poverljivost. To znači da Trudi može da pročita poruku *M*, ali ne može da je promeni zbog postojanja digitalnog potpisa *S* poruke *M*. Trudi je samo pasivan napadač (može da prisluškuje). Vlasnik digitalnog potpisa u ovom scenariju je Alisa. Na drugoj strani, Bob prima poruku *M* i digitalni potpis *S*, a zatim koristi javni ključ $(N, e)$ da bi uspešno verifikovao digitalni potpis i bio siguran da je primljena poruka *M* poslata od Alise.

Kompletan proces generisanja digitalnog potpisa i verifikovanje potpisa (implementacija koja sledi) objasnićemo kroz pet koraka:

1. Pretpostavimo da je Alisa kreirala porudžbenicu za Boba. Alisa prvo generiše heš vrednost porudžbenice preko jednosmerne funkcije *SHA*-1;
2. Alisa sa svojim privatnim ključem šifruje *SHA*-1 heš vrednost. Narudžbenica ostaje u otvorenoj tekstualnoj formi, dok je heš vrednost poslata šifrovano;
3. Bob prima kompletnu poruku koja sadrži narudžbenicu, potpis i specifikaciju korišćenih algoritama;
4. Bob po specifikaciji generiše heš vrednost preko funkcije *SHA*-1 i dobija otisak od 160 bitova;
5. Bob dešifruje potpis Alisinim javnim ključem i na kraju proverava jednakost heš vrednosti. Ukoliko se heš vrednosti poklapaju, Bob je siguran da je narudžbenicu Alisa sastavila, u drugo slučaju otkriće napad;

*JCA* ekstenzija uključuje implementaciju *DSA* algoritma za digitalno potpisivanje, koji obezbeđuje sve mehanizme neophodne za uspešni kompletiranje svih faza digitalnog potpisa. Treba imati na umu da se *DSA* algoritam prema standardima *NIST*-a može samo koristiti za operaciju digitalnog potpisivanja, ali ne operaciju šifrovanja. Za operaciju šifrovanja predviđena je upotreba *RSA* algoritma. Ako bismo poredili brzinu ova dva algoritma, dobili bi rezultat gde je generisanje potpisa preko *RSA* algoritma sporo, a preko *DSA* brzo, dok je u procesu verifikovanja *RSA* brži od *DSA* algoritma. Međutim, ukoliko u praksi ne moramo da vodimo previše računa o utrošku računarske snage, logičan izbor algoritma za digitalni potpis pada na *RSA*. Iz tog razloga primeri koji slede zasnovani su na *RSA* algoritmu.

### 4.3.1 DIGITALNO POTPISIVANJE

Svaka implementacija Java platforme zahteva podršku sledećih algoritama za generisanje digitalnog potpisa:

- *SHA1withDSA*
- *SHA1withRSA*
- *SHA256withRSA*

Sledi primer (Primer 4.6) u kome ćemo pokazati korišćenje Java *API*-ja za generisanje ključeva i digitalnog potpisa. Digitalno ćemo potpisati poruku, zapisati u fajl javni ključ, digitalni potpis i potpisane podatke.

```
public class DigitalniPotpis {

    public static void main(String[] args) throws NoSuchAlgorithmException, NoSu
        //alisa genriše par ključeva
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
        SecureRandom CSPRNG = SecureRandom.getInstance("SHA1PRNG");
        keyGen.initialize(1024, CSPRNG);

        KeyPair pair = keyGen.generateKeyPair();
        PrivateKey privatni = pair.getPrivate();
        PublicKey javni = pair.getPublic();
        //Alisa bira algoritme - izabrala je RSA sa SHA-1
        Signature dsa = Signature.getInstance("SHA1withRSA");
        dsa.initSign(privatni);
        //Alisa učitava poruku koju će da digitalno potpiše
        FileInputStream poruka = new FileInputStream("video.avi");
        BufferedInputStream bufporuka = new BufferedInputStream(poruka);
        byte[] buffer = new byte[1024];
        int len;
        while (bufporuka.available() != 0) {
            len = bufporuka.read(buffer);
            dsa.update(buffer, 0, len);
        }
        bufporuka.close();
        poruka.close();
        //Alisa generiše digitalni potpis pozivajući metod sign()
        byte[] digitalniPotpis = dsa.sign();
        //Alisa šalje potpis Bobu
        sacuvaj(digitalniPotpis, "potpis.sig");
        //Alisa šalje javni ključ Bobu
        byte[] key = javni.getEncoded();
        sacuvaj(key, "javniKljuc.JK");
        System.out.println("Digitalni potpis: "
                +Hex.toHexString(digitalniPotpis));
        System.out.println("Operacija uspešna!");
    }

    public static void sacuvaj(byte podaci[], String nazivFajla) throws IOExcept
        Path lokacija = Paths.get(nazivFajla);
        Files.write(lokacija, podaci, StandardOpenOption.CREATE_NEW);
    }
}
```

*Primer 4.6 Digitalni potpis* – RSA *i* SHA-*1 algoritam*

Programski izlaz:

```
Digitalni potpis: 4ba130f41be0702f2450fd1fa9f1c852617e1cf5bc7a454097f1d7d536
29b4b5e9576e8ba6804f8ec6411f704ec15310611a92471394500da6fc1ccbb6dcfec81b6926
274b2b9b09ba296e0283ca149136315160badd290c8ce7966eedc49d9d5b2b8bbbe6c296d383
d63ceb8036d0776754e1b6b9e4399553f1c3729f11fb2e
Operacija uspešna!
```

U primeru za generisanje digitalnog potpisa, Alisa je prvo generisala par ključeva (javni i privatni). Zatim, inicijalizuje algoritam *SHA1* (za heš vrednost) sa *RSA* (za šifrovanje) za generisanje digitalnog potpisa. Učitava datoteku iterativno u bafer veličine 1024 bajta, gde u svakoj iteraciji poziva metod `update()` (režim za višedelne operacije) nad instancom za operaciju digitalnog potpisivanja. Na kraju, kada je učitana kompletna datoteka, generiše digitalni potpis preko metoda `sign()`.

Posle uspešno generisanog digitalnog potpisa, šalje Bobu potpis, svoj javni ključ i poruku. U programskoj simulaciji, navedeni podaci sačuvani su u datotekama, u folderu glavnog projekta. Generisani digitalni potpis prikazan je u heksadecimalnom formatu kao programski izlaz.

Kao što smo već rekli, algoritmi za generisanje digitalnog potpisa mogu da budu različiti. *NIST* standard predviđa upotrebu *DSA* algoritma sa jednosmernom funkcijom *SHA*-1. *DSA* algoritam koristi *SHA*-1 algoritam za generisanje otiska poruke i deklariše se kao *SHA1withDSA*. U slučaju *RSA*, postoji veći izbor jednosmernih funkcija, takođe one mogu biti deklarisane kao *MD2withRSA*, *MD5withRSA* ili *SHA1withRSA*. Dakle, naziv algoritma mora biti deklarisan. U primeru za generisanje digitalnog potpisa koristili smo klasu Signature, preko čije instance smo imali mogućnost da generišemo i verifikujemo potpis.

### 4.3.2 VERIFIKOVANJE DIGITALNOG POTPISA

U primeru (Primer 4.7) demonstriraćemo upotrebu Java *API*-ja za učitavanje javnog ključa, digitalnog potpisa i potpisanu poruku. Digitalno ćemo potpisati dokument, i zapisati u fajl javni ključ, digitalni potpis i potpisane podatke.

```
public class VDigitalniPotpis {
    public static void main(String[] args) throws NoSuchProviderException, NoSuc
        //Učitavanje javnog ključa
        byte[] kljuc = ucitaj("javniKljuc.JK");
        X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(kljuc);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PublicKey javniKljuc = keyFactory.generatePublic(pubKeySpec);
        //Učitavanje Alisinog potpisa
        byte[] digitalniPotpis = ucitaj("potpis.sig");
        //Inicijalizovanje korišćenog algoritma za digitalni poptis
        Signature potpis = Signature.getInstance("SHA1withRSA");
        potpis.initVerify(javniKljuc);
        //Učitavanje Alisine poruke
        FileInputStream poruka = new FileInputStream("video.avi");
        BufferedInputStream bufPoruka = new BufferedInputStream(poruka);
        byte[] buffer = new byte[1024];
        int len;
        while (bufPoruka.available() != 0) {
            len = bufPoruka.read(buffer);
            potpis.update(buffer, 0, len);
        }
        bufPoruka.close();
        //Bob poredi heš vrednosti
        boolean status = potpis.verify(digitalniPotpis);
        if(status){
            System.out.println("Digitalni potpis: \n"
                +Hex.toHexString(digitalniPotpis));
            System.out.println("Verifikovanje uspešno!");
        }else{
            System.out.println("Verifikovanje nije uspešno!:");
        }
    }

    public static byte[] ucitaj(String nazivFajla) throws IOException {
        Path lokacija = Paths.get(nazivFajla);
        return Files.readAllBytes(lokacija);
    }
}
```

*Primer 4.7 Verifikovanje digitalnog potpisa*

Programski izlaz:

```
Digitalni potpis: 4ba130f41be0702f2450fd1fa9f1c852617e1cf5bc7a454097f1d7d536
29b4b5e9576e8ba6804f8ec6411f704ec15310611a92471394500da6fc1ccbb6dcfec81b6926
274b2b9b09ba296e0283ca149136315160badd290c8ce7966eedc49d9d5b2b8bbbe6c296d383
d63ceb8036d0776754e1b6b9e4399553f1c3729f11fb2e
Verifikovanje uspešno!
```

U primeru za verifikovanje digitalnog potpisa, Bob je učitao Alisin javni ključ i digitalni potpis po specifikaciji za kodovanje javnog ključa (*X.509*). Zatim, inicijalizuje algoritam *SHA1* (za heš vrednost) sa *RSA* (za šifrovanje) prema specifikaciji koju je dobio od Alise. Učitava datoteku (poruku) iterativno u bafer veličine 1024 bajta, gde u svakoj iteraciji poziva metod `update()` (režim za višedelne operacije) nad instancom za verifikovanje digitalnog potpis. Na kraju, kada je učitana kompletna datoteka, verifikuje (upoređuje otiske poruke) digitalni potpis preko metoda `verify()`. Metod verify() vratiće rezultat tačno ukoliko je poruka verifikovana ili netačno u ako nije verifikovana.

Posle verifikovanja digitalnog potpisa Bob može da izvede sledeće zaključke:

- Da je sadržaj Alisine poruke izmenjen;
- Da sadržaj Alisine poruke nije izmenjen;

Generisani digitalni potpis prikazan je u heksadecimalnom formatu kao programski izlaz.

### 4.3.3 SLABOSTI I ALTERNATIVE

Primeri koje smo pokazali za generisanje i verifikovanje digitalnog potpisa, ilustruju upotrebu Java mehanizma za proveru autentičnosti poruke. U realnim situacijama ključevi koji se koriste verovatno će biti već generisani i sačuvani u datoteci.

Potencijalni glavni nedostatak u ovim primerima je da ništa ne garantuje autentičnost javnih ključeva. Primeri koje smo pokazali ispravno verifikuju potpis, ali samo pod uslovom da je isporučen autentičan ključ, a to znači da je javni ključ koji je Bob primio, takođe, potpisan od strane nekog vrhovnog *CA* tela iz *PKI* infrastrukture.

# 5. UPRAVLJANJE KLJUČEVIMA I SERTIFIKATIMA

U ovom poglavlju, diskutovaćemo o mehanizmima za upravljanje sertifikatima i simetričnim ključevima. Pod upravljanjem ključevima, podrazumevamo Java implementaciju skladišta ključeva za različite tipove ključeva (simetrični i asimetrični) i sertifikata, sa obezbeđenom kontrolom pristupa radi zaštite uskladištenih ključeva. Pokazaćemo na koji način se može iskoristiti *KeyTool* alat za generisanje *DSA* sertifikata pogodnih za digitalno potpisivanje i *RSA* sertifikata za šifrovanje ili digitalno potpisivanje.

Upravljanje tajnim ključevima, parom ključeva i sertifikatima, može biti smatrano velikim poduhvatom u nekim situacijama od izuzetne važnosti. Iz tog razloga uvodimo Java `KeyStore` ili interfejs za upravljanje velikim kolekcijama ključeva. Proći ćemo kroz nekoliko primera koda, koji demonstriraju učitavanje i čuvanje sertfikata, kodovanih sa *BER* (engl. *Basic encoding rules*), *DER* (engl. *Distinguished encoding rules*) i *Base64* algoritmom. Takođe, prikazaćemo primer u kome koristimo javni ključ iz sertifikata, za inicijalizovanje *RSA* instance (`Cipher`) za operaciju šifrovanja.

Problem upravljanja ključevima nije jednostavan za rešavanje i očigledno da nije problem koji matematičari rešavaju. Ovaj proces uključuje ljude, a činjenica je da ljudi nisu savršena bića, a nisu ni mašine. Prirodno je da ljudi mogu da izgube ključeve. Šta će se desiti ako neko izgubi tajni ključ? Da li će podaci biti trajno izgubljeni? Da li postoji neki pouzdan mehanizam ili sistem za upravljanje ključevima koji će ključeve držati na bezbednom mestu? Sve su ovo pitanja na koja treba dati odgovore i smatramo da to neće biti jednostavno. Zapravo, u ovom poglavlju se nećemo baviti veličinom kriptološkog ključa ili korišćenim izvorom slučajnosti. Ovo su potpuno nove slabosti koje su prisutne kod svakog kriptosistema i koje je potrebno prevazići.

## 5.1 DIGITALNI SERTIFIKATI

Sertifikati predstavljaju digitalne pasoše izdate od sertifikacionih tela *CA* (engl. *Certificate Authority, CA u daljem tekstu*) raznim učesnicima u komunikaciji, pri tome garantujući njihov identitet i potvrdu istog ostalim učesnicima na Internetu. Korisnici u komunikaciji utvrđuju autentičnost sertifikata proverom da li je sertifikat izdat od

pouzdanog *CA* i ukoliko jeste komunikacija se ostvaruje između odgovarajućih korisnika. Ukoliko sertifikat nije validan komunikacija može biti odbijena.

Upotreba digitalnih sertifikata rešava neke od ključnih problema bezbednosti informacija. Ukoliko nam poruka stiže od korisnika čiji identitet utvrđujemo potpisom treće strane kojoj se veruje, nema razloga da sumnjamo da je takva poruka kompromitovana, odnosno da je stigla od osobe koje je neovlašteno presrela komunikaciju.

Proverom potpisa na sertifikatu istovremeno se utvrđuje i identitet vlasnika odgovarajućeg javnog ili privatnog ključa. Međutim, na ovaj način ne može se utvrditi identitet izdavača sertifikata. Sertifikati su javni. Zajednički format svih sertifikata, definisan je sa standardom *X.509*.

Validnost sertifikata može biti opozvana iz brojnih razloga, a osnovna uloga poverenja u *CA* kompromitovana. Dosadašnja praksa je pokazala da su najčešće kompromitovani sertifikati *CA* izdatih za bankarsko odnosno finansijsko poslovanje, vladine organizacije i pojedine državne bezbednosne agencije. Jednom kompromitovani *CA* veoma teško vraća poverenje korisnika. Značaj kompromitovanih podataka, vrsta klijenata i vreme oporavka ukoliko je moguće, direktno utiče na ugled i poslovno okruženje izdavaoca sertifikata, a šteta može biti trajna. Jedan od primera je dat u analizi kompromitacije sertifikata firme *DigiNotar Certificate Authority*. Najveći problem za korisnike predstavlja nebezbednost u periodu od nastanka kompromitacije *CA* do vremena kada su preduzetim akcijama otklonjeni uzroci. Vreme otkrivanja i rešavanja problema može biti kratko npr. 1 sat, a ponekad i toliko dugo da se meri u nedeljama. Zapaženo je da ovo vreme direktno utiče na posledice primene lažnih sertifikata.

Format *X.509* određuje format zapisa i definiše polja sertifikata. Ovaj standard takođe predviđa načine povlačenje sertifikata kao i algoritme za šifrovanje. Prvi *PKI* projekat je podrazumevao upotrebu upravo *X.509* standarda. Mnoge kompanije u svetu svoje poslovanje temelje na ovom standardu. *VISA* i *MasterCard* kao u svojim elektronskim transakcijama primenjuju *X.509* standard. U nastavku poglavlja dajemo osnovne strukture *X.509* sertifikata verzije 3.

*Tabela 5.1 Atributi X.509 sertifikata*

| ***X.509* ver3: Standardni sertifikat, odobren od strane *ITU*** | |
|---|---|
| Verzija | Verzija standarda *X.509* |
| Serijski broj | *CA* utvrđuje jedinstvenost sertifikata |
| Algoritam potpisivanja | Koji je algoritam |
| Davalac sertifikata | Ime *CA* po *X.500* |
| Period važenja | Početak i kraj |
| Ime korisnika sertifikata | Korisnik za čiji ključ se garantuje |
| Javni ključ | |
| Identifikator davaoca sertifikata | Neobavezno |
| Identifikator davaoca sertifikata | Neobavezno |
| Proširenja | |
| Potpis | Urađen sa privatnim ključem *CA* |

Osnovni zadatak sertifikata je da poveže javni ključ sa podacima kao što su pojedinac, preduzeće, univerzitet i td. Sertifikati nisu tajni. Alisa i Bob u komunikaciji mogu da naprave hiper vezu na matičnim Veb stranama ka svojim sertifikatima.

U današnjim poslovnim okruženjima postoje kompanije koje izdaju sertifikate, kojima se može verovati i koji su podržani od strane Internet pretraživača. Neke od najpoznatijih kompanija iz ove oblasti su: *VeriSign*, *Entrust Authority* i *Comodo inc*.

*PKI* infrastruktura predviđa postojanje samo jednog vrhovnog *CA*. U stvarnom svetu ne postoji niko ko bi mogao da ispuni kriterijume za tako poverljivu ulogu. Iz tog razloga došlo je i do propasti pojma *PKI*, jer ne postoji takav kandidat. Mnoge velike organizacije (*NSA*, *Microsoft*, i td) su se samostalno pokušale nametnuti kao vrhovne, ali ni jednoj to nije pošlo za rukom.

## 5.2 UPRAVLJANJE SA JAVA SKLADIŠTEM KLJUČEVA

*JCA* definiše `KeyStore` klasu za skladištenje i upravljanje simetričnim i asimetričnim ključevima i digitalnim sertifikatima. Klasa je smeštena u paketu `java.security`. Ova klasa implementira tri osnovna tipa interfejsa za `KeyStore.entry`:

- `KeyStore.PrivateKeyEntry` – ovaj tip interfejsa obezbeđuje zaštitu uskladištenog privatnog ključa od neovlašćenog pristupa. Takođe, povezan je sa lancem sertifikata, sa odgovarajućim javnim ključem;
- `KeyStore.SecretKeyEntry` – ovaj tip interfejsa obezbeđuje zaštitu uskladištenog tajnog ključa od neovlašćenog pristupa;
- `KeyStore.TrustedCertificateEntry` – ovaj tip interfejsa drži uskladišteno sertifikat sa javnim ključem od druge strane, strane kojoj se veruje da je sertifikat zaista njen. Ovaj tip interfejsa može biti korišćen za autentifikovanje druge strane;

Svaki unos u `KeyStore` ima svoje „alias“ ili ime preko koga ga je moguće identifikovati određeni entitet. U slučaju privatnih ključeva i njihovih povezanih lanaca sertifikata, autentifikaciju entiteta je moguće obezbediti na više različitih

načina. Na primer, entitet može biti autentifikovan preko *CA* ili preko različitih algoritama sa javnim ključem.

Preko sledeće linije Java koda deklarišemo i inicijalizujemo instancu `KeyStore` skladišta:

```
KeyStore ks = KeyStore.getInstance("JKS");
```

Posle kreiranja instance `KeyStore` možemo da učitamo ključeve koristeći ulazni tok podataka (`FileInputStream`) i lozinku koja štiti ključeve od neovlašćenog pristupa. Sledi fragment nepotpunog koda, koji ilustruje proces pristupa i učitavanja privatnog ključa iz skladišta:

```
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());

char[] lozinka = {'S','i','n','g','i','d','u','n','u','m'};

try (FileInputStream fis = new FileInputStream("nazivSkladista")) {
    ks.load(fis, lozinka);
}
```

Sada, možemo vrlo jednostavno da pročitamo sve ono što se nalazi u skladištu ključeva, da smestimo nove ključeve u skladište ili da izbrišemo neke ključeve.

Takođe, možemo da koristimo istu lozinku da bi zaštitili skladište, ali pored jedne lozinke moguće je koristiti i različite lozinke ili druge parametre zaštite. Sve implementacije Java platforme zahtevaju podršku za `KeyStore` preko standarda *PKCS12*. Java provajder Sun*JCE* obezbeđuje jaku zaštitu ključeva u skladištu, tako što implementira *TDES* algoritam sa *PBE* za šifrovanje.

U sledećem primeru (Primer 5.1) pokazaćemo jedan način za kreiranje novog skladišta ključeva u koga ćemo smestiti lozinku „Singidunum“.

```
public class SkladisteKljuceva {
    public static void main(String[] args) throws KeyStoreException, IOExceptior
        //Kreiranje novog praznog skladišta ključeva - Java KeyStore
        KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType()) ;
        ks.load(null,"".toCharArray());
        //Otvaranje izlaznog toka podataka ka lokaciji skladišta
        FileOutputStream fos =
                new FileOutputStream(new File("SkladišteKljučeva"));
        //Postavljanje lozinke za pristup skladištu
        char[] lozinka = {'S','i','n','g','i','d','u','n','u','m'};
        ks.store(fos, lozinka);
        //Čišćenje niza u kome je lozinka bila
        Arrays.fill(lozinka, '\u0000' ) ;
        System.out.println("Kreirano je prazno skladište!");
        System.out.println("Tip skladišta: "+ks.getType());
    }
}
```

*Primer 5.1 Kreiranje praznog* JKS *skladišta*

Programski izlaz:

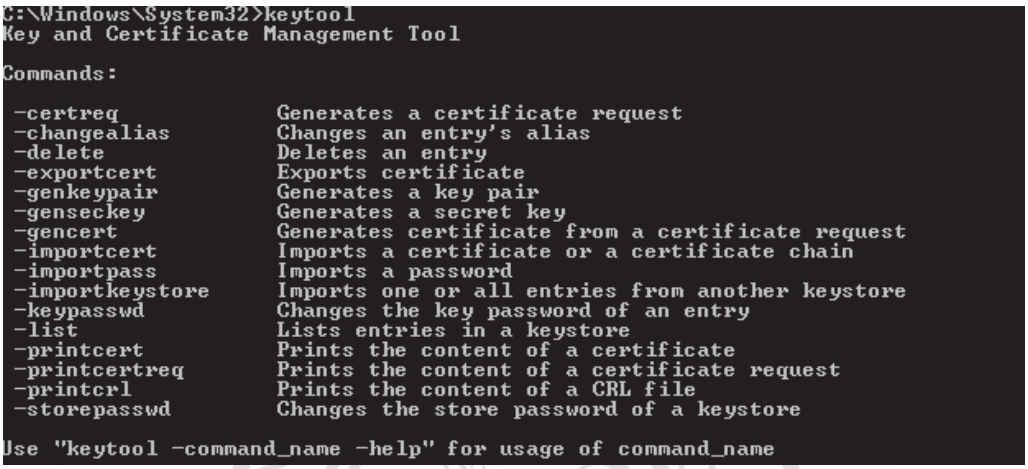
```
Kreirano je prazno skladište!
Tip skladišta: jks
```

Kao što vidimo, u primeru iznad kreirano je novo skladište ključeva - *JKS*, postavljena je lozinka za budući pristup skladištu i određena lokacija skladišta (datoteka).

### 5.2.1 GENERISANJE SERTIFIKATA

Java platforma u sebi sadrži jednostavan alat – *KeyTool*. Alat se koristi za upravljanje sa skladištem ključeva i daje mogućnost za generisanje novih različitih tipova sertifikata. U praksi se često podrazumeva da sertifikati budu generisani nezavisno od aplikacije, a to znači da sertifikati neće biti generisani dinamički od strane aplikacije. Kod dinamički generisanih sertifikata glavni problem bi bilo poverenje korisnika, a time bi sertifikati izgubili na značaju. Poslovna okruženja zahtevaju da sertifikati budu potpisani od nekog vrhovnog *CA*, ali za svrhu razvoja i eksperimentisanja ovi sertifikati su više nego dovoljni.

*KeyTool* se jednostavno pokreće. Potrebno je pokrenuti konzolni prozor na operativnom sistemu (ponekad je potrebno podesiti sistemsku putanju da bi komanda bila prepoznata) i otkucati komandu *KeyTool*. Posle čega će se pojaviti prozor prikazan na sledećoj slici.




*Slika 5.1 KeyTool konzolni interfejs*

Na slici (Slika 5.1), su prikazane funkcije dostupne preko *KeyTool*-a. Ove funkcije obezbeđuju sve neophodne funkcionalnosti za upravljanje skladištima ključeva i digitalnim sertifikatima.

Navešćemo neke najbitnije grupe komande za rukovanje *KeyTool* alatom:

- Generisanje Java skladišta, ključeva i sertifikata,
- Uvoz sertifikata,
- Izvoz sertifikata,
- Prikazivanje sertifikata,
- Brisanje sertifikata,
- Promena lozinki;

Pre nego što demonstriramo kroz više primera upotrebu ovog alata, kreiraćemo jedan sertifikat. To ćemo uraditi jednostavno pozivajući komandu –genkey. Inicijalno komanda bez specifikacije algoritma će generisati *DSA* sertifikat koristan za digitalni

potpis, ali ne i za šifrovanje. Kasnije, preko naziva alijasa ćemo preuzeti sertifikat iz Java skladišta. Preuzimanje možemo obaviti preko *Keytool*-a ili programski.

**-v –** prikaz povratnih informacija nakon izvršenih komandi

**keytool -genkey -alias** *DSAsertifikat* **-v -keystore** /*skladiste.keystore*

```
C:\Windows\system32>keytool -genkey -alias DSAsertifikat -v -keystore /skladiste
.keystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Sasa Adamovic
What is the name of your organizational unit?
  [Unknown]:  Fakultet za informatiku i racunarstvo
What is the name of your organization?
  [Unknown]:  Singidunum Univerzitet
What is the name of your City or Locality?
  [Unknown]:  Beograd
What is the name of your State or Province?
  [Unknown]:  Srbija
What is the two-letter country code for this unit?
  [Unknown]:  RS
Is CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum Univ
erzitet, L=Beograd, ST=Srbija, C=RS correct?
  [no]:  yes

Generating 1,024 bit DSA key pair and self-signed certificate (SHA1withDSA) with
 a validity of 90 days
        for: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singi
dunum Univerzitet, L=Beograd, ST=Srbija, C=RS
Enter key password for <DSAsertifikat>
        (RETURN if same as keystore password):
[Storing /skladiste.keystore]
```

*Primer 5.2 Generisanje* DSA *sertifikata*

Sada ćemo napraviti jedan dodatni sertifikat za *RSA* algoritamsku specifikaciju. *RSA* algoritam pored operacije potpisivanja, podržava i šifrovanje. Konzolna komanda će biti slična prethodnoj. Dodaćemo još jednu komandu za definisanje algoritma i promenićemo naziv alijasa.

**keytool -genkey -alias** *RSAsertifikat* **-v -keyalg** *RSA* **-keystore** /*skladiste.keystore*

Novi *RSA* sertifikat će biti generisan u postojećem skladištu ključeva i sertifikata, samo pod nazivom alijas-a *RSAsertifikat*. Veličina *RSA* ključeva koja je generisana je 2048 bitova, samopotpisani sertifikat (*SHA256withRSA*) na 90 dana.

```
C:\Windows\system32>keytool -genkey -alias RSAsertifikat -v -keyalg RSA -keystor
e /skladiste.keystore
Enter keystore password:
What is your first and last name?
  [Unknown]:  Sasa Adamovic
What is the name of your organizational unit?
  [Unknown]:  Fakultet za informatiku i racunarstvo
What is the name of your organization?
  [Unknown]:  Singidunum Univerzitet
What is the name of your City or Locality?
  [Unknown]:  Beograd
What is the name of your State or Province?
  [Unknown]:  Srbija
What is the two-letter country code for this unit?
  [Unknown]:  RS
Is CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum Univ
erzitet, L=Beograd, ST=Srbija, C=RS correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) wi
th a validity of 90 days
        for: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singi
dunum Univerzitet, L=Beograd, ST=Srbija, C=RS
Enter key password for <RSAsertifikat>
        (RETURN if same as keystore password):
[Storing /skladiste.keystore]
```

*Primer 5.3 Generisanje* RSA *sertifikata*

U ovom momentu u našem skladištu ključeva „skladiste.keystore“ imamo dva sertifikata. *DSA* sertifikat veličine 1024 bita i *RSA* sertifikat veličine 2048 bitova. Implementacija skladišta ključeva podržava skladište dva tipa objekata (ključevi i sertifikati) i sertifikate od poverenja potpisane od strane *CA*.

Takođe, treba imati na umu da pored bezbednog skladištenja ključeva i sertifikata, uloga skladišta ključeva (*KeyStore*) nije da verifikuje autentičnost sertifikata. Unutar njega ne postoji algoritam koji će obezbediti takvu vrstu funkcionalnosti.

U nastavku, sledi primer (Primer 5.4) Java koda u kome ćemo prikazati listu svih entiteta i njihove alijas-e u skladištu „skladiste.keystore“. Da bismo pristupili skladištu ključeva, moramo da koristimo lozinku koju smo postavili u trenutku kreiranja skladišta. Naša lozinka je „Singidunum“. Postoji mogućnost da nezavisno od lozinke skladišta kreiramo različite lozinke za entitete unutar skladišta. U našem slučaju, mi smo koristili istu lozinku za sve entitete i skladište. Kada smo generisali entitete (*DSA* i *RSA* sertifikat) u konzolnom interfejsu, na kraju pored zahteva da unesemo lozinku entiteta dobijamo i drugu mogućnost u vidu poruke *<RETURN if same as keystore password>*, tada je dovoljno samo pritisnuti taster *enter* za korišćenje iste lozinke za sve.

```
public class PregledSkladista {

    public static void main(String[] args) throws KeyStoreException, FileNotFour
        String formatSkladista = Security.getProperty("keystore.type");
        KeyStore skladiste = KeyStore.getInstance(formatSkladista);
        FileInputStream fis = new FileInputStream("skladiste.keystore");
        char[] lozinka =
                new char[]{'S', 'i', 'n', 'g', 'i', 'd', 'u', 'n', 'u', 'm'};
        skladiste.load(fis, lozinka);
        Arrays.fill(lozinka, '\u0000');

        for (Enumeration e = skladiste.aliases(); e.hasMoreElements();){
            String alijas = (String) e.nextElement();
            System.out.println("Pronađen alijas: " + alijas);
        }
    }
}
```

*Primer 5.4 Programski pregled skladišta ključeva i sertifikata*

Programski izlaz:

```
Pronađen alijas: rsasertifikat
Pronađen alijas: dsasertifikat
```

Na osnovu programskog izlaza, vidimo da se u skladištu „skladiste.keystore“, nalaze dva sertifikata.

### 5.2.2 SKLADIŠTA ZA SIMETRIČNE KLJUČEVE

U ovom delu videćemo kakve mogućnosti postoje za skladištenje tajnih ključeva. Možemo izabrati dva tipa `KeyStore` skladišta. Prvi tip skladišta je *JCEKS* i taj tip ćemo koristiti u prvom prime koji sled. Drugi tip skladišta je *BKS*. Ovaj tip skladišta se odnosi na drugi provajder, konkretno radi se o provajderu *BouncyCaste,* provajder o kome je bilo više reči u prethodnim poglavljima.

U primeru koji sledi, kreirali smo novo skladište tipa *JCEKS*. Ovaj tip obezbeđuje podršku za skladištenje tajnih ključeva. Generisan je AES ključ i pod alijasom „AEStajniKljuc“ skladišten. Radi provere uspešnosti skladištenja tajnog ključa, prikazaćemo i pregled skladišta „skladste.JCEKSkeystore“.

```
public class SkladistenjeTajnihKjuceva {

    public static void main(String[] args) throws FileNotFoundException, IOExcep
        //Deklarisanje novog skladišta
        File fajl = new File("skladiste.JCEKSkeystore");
        //Pristupna lozinka
        char[] lozinka
                = new char[]{'S', 'i', 'n', 'g', 'i', 'd', 'u', 'n', 'u', 'm'};
        //Definisanje tipa skladišta
        KeyStore skladiste = KeyStore.getInstance("JCEKS");
        skladiste.load(null, new char[0]);
        //Generisanje tajnog ključa
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        SecretKey kljuc = kg.generateKey();
        byte[] aesKljuc = kljuc.getEncoded();
        //Unos novog entiteta za alijas - AEStajniKljuc
        skladiste.setKeyEntry("AEStajniKljuc", kljuc, lozinka, null);
        FileOutputStream fos = new FileOutputStream(fajl);
        skladiste.store(fos, lozinka);
        //Trajno brisanje lozinke skladišta
        Arrays.fill(lozinka, '\u0000');
        System.out.println("AES ključ je sačuvan u skladištu ključeva!");
    }
}
```

*Primer 5.5 Skladištenje tajnih ključeva (*AES *ključ)*

Programski izlaz:

```
AES ključ je sačuvan u skladištu ključeva!

Pronađen alijas: aestajnikljuc
```

U primeru (Primer 5.5), prva linija programskog izlaza daje informaciju o uspešnosti skladištenja tajnog ključa, dok drugi programski izlaz nastaje preko primera (Primer 5.4) i daje uvid u listu entiteta novog skladišta. U skladištu se nalazi jedan entitet – „aestajnikljuc“.

Zahtevana je mala modifikacija u Java kodu za pregled entiteta. Naime, potrebno je definisati tip skladišta (*JCEKS*) umesto "keystore.type", da bismo mogli da učitamo sadržaj novog tipa skladišta.

### 5.2.3 UČITAVANJE SIMETRIČNIH KLJUČEVA IZ SKLADIŠTA

U primeru (Primer 5.6), pokazaćemo na koji način možemo da pristupimo skladištu ključeva preko korisnički definisane lozinke - PBE. U skladištu ključeva nalazi se jedan entitet – simetrični ključ, koji je potrebno preuzeti iz skladišta i pripremiti ga za dalju upotrebu u *AES* algoritmu.

Pozivom metoda `getKey()` nad instancom `skladiste`, kao rezultat dobijamo novu instancu klase `Key` iz koje preko metoda `getEncoded()` dobijamo niz bajtova, koji reprezentuju naš *AES* ključ uskladišten u prethodnom primeru .

```
public class UzimanjeTajnogKljuca {
    public static void main(String[] args) throws KeyStoreException, FileNotFour
        //Pristupna lozinka od skladišta
        char[] lozinka =
                new char[]{'S', 'i', 'n', 'g', 'i', 'd', 'u', 'n', 'u', 'm'};
        //Učitavanje skladišta
        KeyStore skladiste = KeyStore.getInstance("JCEKS");
        FileInputStream fis = new FileInputStream("skladiste.JCEKSkeystore");
        skladiste.load(fis, lozinka);
        //Uzimanje tajnog ključa preko alijasa
        Key kljuc = skladiste.getKey("AEStajniKljuc", lozinka);
        byte[] aesKljuc = kljuc.getEncoded();
        //Prikaz tajnog ključa
        System.out.println("Ključ je uspešno preuzet iz skladišta:");
        for(int i:aesKljuc){
            System.out.print(i+", ");
        }
        Arrays.fill(lozinka, '\u0000' ) ;
        System.out.println("\nKljuč (bit): "+(aesKljuc.length*8)+" bitova.");
    }
}
```

*Primer 5.6 Uzimanje simetričnog ključa iz skladišta*

Programski izlaz:

```
Ključ je uspešno preuzet iz skladišta:
-93, -84, -78, -75, 109, -93, 41, 109, 64, 127, 117, 98, 103, -75, 105, -84,
Ključ (bit): 128 bitova.
```

Programski izlaz prikazuje bajtove ključa i veličinu u bitovima. Kao što vidimo dužina ključa je 128 bitova.

## 5.3 UPRAVLJANJE SERTIFIKATIMA PO X.509 STANDARDU

Postoje dva osnovna formata za izradu digitalnih sertifikata. To je mobilni format i format za transport. Prvi format podrazumeva kodovanje preko niza istaknutih pravila prema *DER* formata (binarna forma), a drugi format podrazumeva kodovanje sertifikata koji su već u *DER* formatu sa *Base64* algoritmom. Postoje i drugi pristupi za čitanje, kao što je pristup preko `CertificateFactory` klase u Javi.

U praksi, retko kad sami generišemo sertifikat preko *KeyTool*-a, najčešće dobijemo sertifikat preko e-mejla, diska ili nekog drugog prenosnog medija. Pre nego što pokažemo primere koda koji demonstriraju čitanje oba tipa sertifikata, treba nam skladište sertifikata sa dva entiteta koje smo već kreirali (DSAsertifikat i RSAsertifikat). Zatim, uz pomoć *KeyTool*-a izvešćemo sertifikate na fajl sistem.

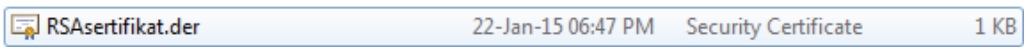
U primeru (Primer 5.7), izvešćemo sertifikat za alias „RSAsertifikat“ koristeći *DER* kodovanje i sertifikat za alias „DSAsertifikat“, koristeći *Base64* kodovanje. Posle izvoza oba sertifikata, videćemo programski, kroz primere koji slede kako možemo da čitamo svaki od ovih formata kodovanja.

**keytool -export -alias** *RSAsertifikat* **-keystore** */skladiste.keystore* **-file** */RSAsertifikat.der*

```
C:\Windows\system32>keytool -export -alias RSAsertifikat -keystore /skladiste.ke
ystore -file /RSAsertifikat.der
Enter keystore password:
Certificate stored in file </RSAsertifikat.der>
```

*Primer 5.7 Izvoz* RSA *sertifikata u* DER *formatu*

Programski izlaz:

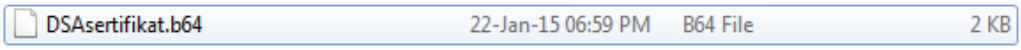


**keytool -export -alias** *DSAsertifikat* **-keystore** */skladiste.keystore* **-rfc -file** */DSAsertifikat.b64*

```
C:\Windows\system32>keytool -export -alias DSAsertifikat -keystore /skladiste.ke
ystore -rfc -file /DSAsertifikat.b64
Enter keystore password:
Certificate stored in file </DSAsertifikat.b64>
```

*Primer 5.8 Izvoz* DSA *sertifikata u Base64 formatu*

Programski izlaz:




Oba izvezena sertifikata nalaze se u *root*-u diska, kao i skladište „skladiste.keystore“. Da ne zaboravimo, koristimo lozinku „Singidunum“ za pristup entitetima unutar skladišta.

U prvom primeru (Primer 5.9) koji sledi, demonstriraćemo čitanje sertifikata kodovanog u *DER* formatu.

```
public class DERcitanjeSertifikata {
    public static void main(String[] args) throws CertificateException, FileNotF
       //Kreiranje instance fabrika po X.509 specifikaciji
       CertificateFactory fabrika = CertificateFactory.getInstance("X.509");
       //Učitavanje sertifikata u DER formatu - RSAsertifikat
       File fajl = new File("RSAsertifikat.der");
       FileInputStream fis = new FileInputStream(fajl);
       //Generisanje
       Certificate sertifikat = fabrika.generateCertificate(fis);
       System.out.println("Sertifikat učitan: \n" + sertifikat.toString());
       fis.close();
    }
}
```

*Primer 5.9 Programsko čitanje sertifikata* RSA *sertifikat u* DER *formatu*

Programski izlaz je kompletan sertifikat sa svim svojim atributima.

Programski izlaz:

```
  Version: V3
  Subject: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum Univ
erzitet, L=Beograd, ST=Srbija, C=RS
  Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

  Key:  Sun RSA public key, 2048 bits
  modulus: 2019030004250977514994752037325212938405300608659593516696721092683492115418
4135825352406840586435969677264700997252465966340968084675936148010762816929856159028323
6540317329344133926142285778860531776643555677594461607507953287238677909420366455524939
5599543851626345471855795553528744242845935844638197076515999224465508106401722251673502
4747703730822669014595449623999451394410820035557324218291533638059212777501575042879784
9737337476270572710059050912515389576143667741630144254190585531753251569101030619152802
1212822843988948345167493711540611679258849066953381020249974052623419278827079847356403
951615965043
  public exponent: 65537
  Validity: [From: Thu Jan 22 14:55:01 CET 2015,
              To: Wed Apr 22 15:55:01 CEST 2015]
  Issuer: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum Unive
rzitet, L=Beograd, ST=Srbija, C=RS
  SerialNumber: [    2708b0ae]


Certificate Extensions: 1
[1]: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 79 0F 54 79 16 86 53 00   11 84 10 AF 14 36 BD 54  y.Ty..S......6.T
0010: 8F FA AF 9B                                        ....
]
]

]
  Algorithm: [SHA256withRSA]
  Signature:
0000: 36 DB CB 86 40 47 78 FB   46 42 F7 7B 08 26 39 51  6...@Gx.FB...&9Q
0010: 32 CE A9 B7 69 31 00 8B   BF 18 BF DC AF EA 33 5C  2...i1........3\
0020: 15 ED 00 47 C3 7F 65 40   F3 72 4A 8E 2C BE 5C 35  ...G..e@.rJ.,.\5
0030: FE 11 22 5D 0F 32 F7 F3   98 5A F9 B2 CE BB 08 57  .."].2...Z.....W
0040: 3C B1 F5 4F 6F 86 22 BE   F9 5A 9E FB A0 18 FF 5F  <..Oo."..Z....._
0050: 79 4B 7A 7F 96 28 D0 10   EE A7 A0 39 4D 9F EF 1E  yKz..(.....9M...
0060: 01 2F B9 F9 2E CF 4E 32   62 36 05 C0 EA 42 A5 1F  ./....N2b6...B..
0070: DA 09 B4 78 F7 8E 93 18   C7 7C 94 DD E4 B4 B9 1A  ...x............
0080: 70 61 EC 01 36 2E 49 6D   73 9D 94 11 50 2C F9 B5  pa..6.Ims...P,..
0090: 1F 9F E2 02 35 05 A0 36   1E 88 DE 52 00 6D 15 3D  ....5..6...R.m.=
00A0: F5 D4 31 A6 D7 03 1C 4D   44 4C 22 2D EE 42 15 2C  ..1....MDL"-.B.,
00B0: 31 B1 75 E8 1F 64 50 12   3F 60 3A F6 A0 28 67 9D  1.u..dP.?`:..(g.
00C0: 2D 5B 2B 01 47 D8 C9 A5   8D 0F 21 D3 F8 62 E1 D9  -[+.G.....!..b..
00D0: E1 E1 7E AE D8 4D C7 34   A0 70 D2 84 7D B1 89 41  .....M.4.p.....A
00E0: 42 71 3D E5 58 1C 5F 43   EA 01 EE 15 C4 2C AA 9E  Bq=.X._C.....,..
00F0: EB 82 E9 DF E9 AE A0 B4   80 BA 53 F2 8F 72 02 CF  ..........S..r..
```

Nakon što je sertifikat izvezen u *DER* formatu, čitanje u Java programu i dalji rad sa njim je vrlo jednostavan. Izvezeni *RSA* i *DSA* sertifikat su reprezentovani kao *X.509*

sertifikati preko `CertificateFactory` klase. U drugom primeru (Primer 5.10), demonstriraćemo čitanje sertifikata kodovanog u Base64 formatu.

```
public class Base64citanjesertifikata {
    public static void main(String[] args) throws FileNotFoundException, Certif:
       //Kreiranje instance fabrika po X.509 specifikaciji
       CertificateFactory fabrika = CertificateFactory.getInstance("X.509");
       //Učitavanje sertifikata u Base64 formatu - DSAsertifikat
       Path lokacija = Paths.get("DSAsertifikat.b64");
       byte dsasertifikatB64 [] = Files.readAllBytes(lokacija);
       ByteArrayInputStream bais = new ByteArrayInputStream(dsasertifikatB64);
       //Generisanje
       Certificate sertifikat = fabrika.generateCertificate(bais);
       System.out.println("Sertifikat učitan: \n" + sertifikat.toString());
    }
}
```

*Primer 5.10 Programsko čitanje sertifikata* DSA *sertifikat u Base64 formatu*

Programski izlaz:

```
Sertifikat učitan:
[
[
  Version: V3
  Subject: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum
Univerzitet, L=Beograd, ST=Srbija, C=RS
  Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

  Key:  Sun DSA Public Key
    Parameters:DSA
        p:     fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b65126
69
    455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
    6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
        q:     9760508f 15230bcc b292b982 a2eb840b f0581cf5
        g:     f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d0782
67
    5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
    3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
    cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

  y:
    505235ee 53125509 887f01aa 9e4d48b3 fc3eb9f5 21425e8d 9555d027 7c59d73e
    5959fa3a d3ede17d c3d4048d 5e8cb453 5a616003 a534ad0f ef2bab8d 56524cd5
    6fad9bae 60ffed52 4bcec959 f6eeab41 1276e17e 618e3cca 549c8f6f f2fa161c
    99968061 f39b8de3 e6080511 92e4f32c 9e80b99b a1ab781a c723aba7 478eeef9

  Validity: [From: Thu Jan 22 14:52:22 CET 2015,
              To: Wed Apr 22 15:52:22 CEST 2015]
  Issuer: CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum U
niverzitet, L=Beograd, ST=Srbija, C=RS
  SerialNumber: [    7f5df54e]

Certificate Extensions: 1
[1]: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: AB 77 4D 6E 70 92 3E 4A   17 FB 62 3F 8C 7C EE 1D  .wMnp.>J..b?....
0010: 4D 62 12 E2                                        Mb..
]
]

]
  Algorithm: [SHA1withDSA]
  Signature:
0000: 30 2C 02 14 69 CC F0 64   22 08 F7 9D 08 42 7F B7  0,..i..d"....B..
0010: F1 C6 09 1B 0A A4 4E 92   02 14 19 AF BD AB BE AC  ......N.........
0020: 09 39 EF 5E AD 76 17 67   46 6B E0 37 00 DD        .9.^.v.gFk.7..
```

### 5.3.1 KREIRANJE ZAHTEVA ZA IZDAVANJE SERTIFIKATA - *CSR*

Zahtev za izdavanje sertifikata *CSR* (engl. *Certificate Signature Request*), predstavlja standardnu formu (po *PKCS#10* standardu) koja se koristi za slanje zahteva za izdavanje potpisanog sertifikata od strane *CA* – generisanje sertifikata od poverenja.

Pre nego što dobijemo digitalno potpisan sertifikat od strane *CA*, potrebno je da saznamo koje algoritme naš *CA* podržava. Većina će podržati *RSA* za digitalno potpisivanje, dok su drugi namenjeni DSA algoritmu

Koristeći *KeyTool*, preko komande **–genkey**, generisaćemo privatni i javni par ključeva pod alijasom „sertifikatParKljuceva“. Potrebno je popuniti sva tražena polja, kao što je prikazano na konzolnom interfejsu ispod.

**keytool -genkey –keyalg** *RSA* **-keysize** *2048* **-alias** *sertifikatParKljuceva*

```
C:\Windows\system32>keytool -genkey -keyalg RSA -keysize 2048 -alias sertifikatP
arKljuceva
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Sasa Adamovic
What is the name of your organizational unit?
  [Unknown]:  Fakultet za informatiku i racunarstvo
What is the name of your organization?
  [Unknown]:  Singidunum Univerzitet
What is the name of your City or Locality?
  [Unknown]:  Beograd
What is the name of your State or Province?
  [Unknown]:  Srbija
What is the two-letter country code for this unit?
  [Unknown]:  RS
Is CN=Sasa Adamovic, OU=Fakultet za informatiku i racunarstvo, O=Singidunum Univ
erzitet, L=Beograd, ST=Srbija, C=RS correct?
  [no]:  yes

Enter key password for <sertifikatParKljuceva>
        (RETURN if same as keystore password):
```

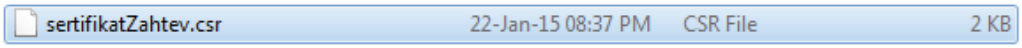
*Primer 5.11 Generisanje privatnog i javnog para ključeva*

Kada imamo generisan par ključeva, možemo ponovo preko *KeyTool*-a da pozovemo komandu **–certreq** za generisanje *CSR* forme zahteva:

```
C:\Windows\system32>keytool -certreq -alias sertifikatParKljuceva -file /sertifi
katZahtev.csr
Enter keystore password:
```

*Primer 5.12 Kreiranje zahteva za digitalni potpis*

Programski izlaz:




Kao što vidimo, programski izlaz je novi fajl sa ekstenzijom *.CSR*. Ne zaboravite, da digitalni sertifikati sadrže javni ključ i meta podatke o vlasniku. Ni u jednom trenutku ne treba distribuirati privatni ključ, čak ni *CA* ne treba da zna privatni ključ. Generisani par ključeva treba čuvati, ukoliko izgubimo privatni ključ, sertifikat postaje bezvredan i više nije moguće generisati digitalne potpise ili obavljati operaciju šifrovanja sa javnim ključem. Operacija digitalnog potpisivanja zahteva privatni ključ, a ne javni sertifikat. Svakako, postoji mnogo razloga za rad sa sertifikatima od poverenja. Na primer, možete slati bilo koja dokumenta digitalno potpisana vašim prijateljima ili poslovnim saradnicima.

Nakon kreiranog *CSR* fajla, sledeći korak je slanje *CSR* forme nekom *CA* kojem je potrebno neko vreme da verifikuje sve navedene meta podatke. Ukoliko su svi meta podaci verifikovani (geolokacija, adresa, ime i prezime, Veb adresa i td.), *CA* izdaje digitalni potpis za traženi sertifikat. Pored potpisa, sertifikat je formatiran po standardu *PKCS#7*, što znači da u sebi sadrži kompletan sertifikacioni lanac koji je potpisao. Na kraju, u zavisnosti kako *CA* funkcioniše, vratiće sertifikat kodovan u *DER* formatu ili Base64.

Kada od *CA* dobijemo nazad sertifikat od poverenja, tada možemo javni sertifikat distribuirati mreži neograničene brojnosti poslovnih partnera.

### 5.3.2 LISTA POVUČENIH SERTIFIKATA – *CRL*

Lista povučenih sertifikata *CRL* (engl. *Certificate Revocation List*) je lista izdata i elektronski potpisana od strane CA koja uključuje serijske brojeve povučenih sertifikata i tačno vreme kada su povučeni iz upotrebe. Dobijena lista se mora koristiti

od strane trećih strana uvek kada treba proveriti validnost sertifikata i/ili verifikaciju digitalnog potpisa.

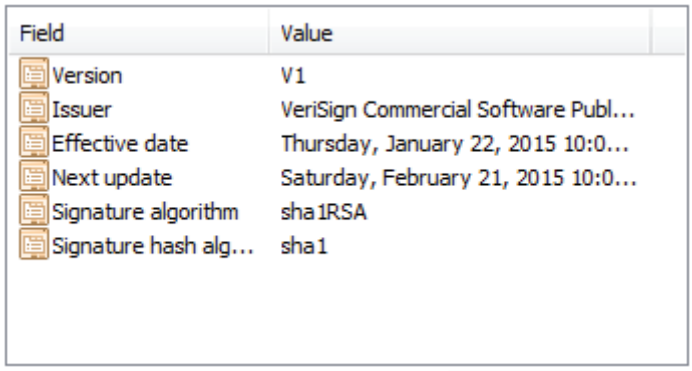
Povlačenje ili suspenzija sertifikata se radi u sledećim slučajevima:

- na zahtev vlasnika sertifikata,
    - neautorizovano objavljivanje,
    - kompromitovanje privatnog ključa;
- na zahtev druge institucije ukoliko smatra da u sertifikatu neki podaci nisu tačni;
    - korisnik ne ispunjava navedene materijalne i nematerijalne obaveze,
    - promena meta podataka u sertifikatu,
    - kompromitovanje *CA;*

Takođe, vreme povlačenja sertifikata, period operativnog rada datog sertifikata se istovremeno smatra završenim. Sertifikat koji je dospeo na *CRL* listu, može biti trajno povučeni iz upotrebe ili privremeno suspendovan (rešavanje manjeg problema zbog kojeg je sertifikat povučen iz upotrebe).

Veliki nedostatak klasa iz domena bezbednosnih klasa u Javi nema implementiranu mogućnost automatskog preuzimanja *CRL* listi (na osnovu *PKCS#7* formata). Takođe, mnoge implementacije *CRL* listi mogu biti ponekad nepouzdane. Problem može da usledi zbog loše komunikacije koja može usporiti rad čitavog sistema prilikom preuzimanja listi od *CA*.

U Javi je moguće analizirati *CRL* listu sa povučenim sertifikatima preko klase `X509CRLEntry`. Da bismo demonstrirali funkcionalnosti ove klase, od organizacije *Verisign* (*CA*), sa adrese crl.verisign.com/Class3SoftwarePublishers.crl, preuzet je fajl „Class3SoftwarePublishers.crl“ sa povučenim sertifikatima.




*Slika 5.2 Informacije o* CRL *listi*

Svaka *CRL* lista mora da poseduje informacije o izdavaču liste, datume, algoritmu koji je korišćen za digitalni potpis i digitalni potpis preko koga je moguće verifikovati validnost liste. Prikazane detaljne informacije sa slike (Slika 5.2), odnose se na *CRL* listu „Class3SoftwarePublishers.crl“ koju je izdala organizacija *Verisign*.

U primeru (Primer 5.13), demonstriraćemo upotrebu Java klase za parsiranje CRL liste.

```
public class PregledCRLliste {
    public static void main(String[] args) throws FileNotFoundException, Certif
        FileInputStream fis =
                new FileInputStream("Class3SoftwarePublishers.crl");
        CertificateFactory fabrika = CertificateFactory.getInstance("X.509");
        X509CRL crl = (X509CRL) fabrika.generateCRL(fis);
        fis.close();
        Set crlSet = crl.getRevokedCertificates();
        Iterator it = crlSet.iterator();
        while (it.hasNext()) {
            X509CRLEntry entry = (X509CRLEntry) it.next();
            System.out.print("Serijski broj sertifikata: ");
            System.out.println(entry.getSerialNumber());
            System.out.print(" je povučen ");
            System.out.println(entry.getRevocationDate());
        }
    }
}
```

*Primer 5.13* CRL *lista sa opozvanim sertifikatima*

Programski izlaz:

```
Serijski broj sertifikata: 188879508029463283546619938014477055
 je povučen Thu Feb 28 06:18:24 CET 2002
Serijski broj sertifikata: 150951223030292553652913388705138684 2
 je povučen Mon Apr 15 17:40:24 CEST 2002
Serijski broj sertifikata: 201242033142477865480596630283905110 7
 je povučen Thu May 09 15:57:58 CEST 2002
Serijski broj sertifikata: 247598306180843452446677710468029044 6
 je povučen Thu Apr 18 19:46:14 CEST 2002
Serijski broj sertifikata: 279683650075590701395167872299037564 3
 je povučen Fri Jul 19 23:28:31 CEST 2002
Serijski broj sertifikata: 305778858908315939776315167567881207 6
 je povučen Tue Feb 12 07:16:50 CET 2002
Serijski broj sertifikata: 312410813145391337179955499363244112 0
 je povučen Thu Jun 19 10:04:47 CEST 2003
Serijski broj sertifikata: 430054127931136978367335986744220662 9
 je povučen Wed Apr 24 21:47:02 CEST 2002
Serijski broj sertifikata: 439481236319415167157243566010962241 1
 je povučen Thu Sep 26 23:51:51 CEST 2002
Serijski broj sertifikata: 503476110802269628600647921477679079 6
 je povučen Thu Apr 18 15:49:22 CEST 2002
```

U praksi, kada je *CRL* lista učitana u memoriju, proverava se validnost sertifikata preko metoda `isRevoked()`, koji vraća rezultat `true` ako je sertifikat na listi ili `false` ukoliko sertifikat nije pronađen. Pretraga se radi po rednom broju sertifikata.

Pored ovog metoda provere, moguće je koristiti *OCSP* (engl. *Online Certificate Status Protocol*) protokol za dobijanje statusa preko Interneta o poništenju *X.509* sertifikata. Ovaj protokol je opisan u dokumentu - *RFC* 6960. Ovaj protokol je nastao kao alternativa *CRL* listi, zbog određenih problema u korišćenju *CRL* listi na infrastrukturi javni ključeva – *PKI*. Poruke koje se razmenjuju preko *OCSP*-a su kodovane u *ASN*.1 formatu i obično se razmenjuju preko *HTTP*-a.

Java provajder *BouncyCastle* podržava *OCSP* protokol, dok mnogi drugi provajderi Java kriptografske arhitekture ne podržavaju.

### 5.3.3 ŠIFROVANJE SA DIGITALNIM SERTIFIKATIMA

Nakon razumevanja procesa za generisanje sertifikata, procesa za kreiranje sertifikata od poverenja (*CSR*), sledi jedan primer (Primer 5.14) u kom ćemo demonstrirati upotrebu digitalnog sertifikata za operaciju šifrovanja. Još jednom napominjemo da sertifikat samo obuhvata javni ključ i u ovom slučaju `Cipher` instancu inicijalizovanu preko preopterećenog metoda `init()` za argument – sertifikat.

```
public class DigitalniSertifikatSifrovanje {
    public static void main(String[] args) throws CertificateException, FileNotF
       //Učitavanje sertifikata
       CertificateFactory factory = CertificateFactory.getInstance("X.509");
       FileInputStream fis = new FileInputStream(new File("RSAsertifikat.der"));
       Certificate sertifikat = factory.generateCertificate(fis);
       System.out.println("Sertifikat je učitan u memoriju.");
       fis.close();
       //Inicijalizovanje šifre - RSA/ECB/PKCS1Padding
       Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding") ;
       sifra.init(Cipher.ENCRYPT_MODE, sertifikat);
       String otvorenaPoruka = "Praktikum - Zaštita informacionih sistema";
       //Operacija šifrovanja
       byte sifrat[] = sifra.doFinal(otvorenaPoruka.getBytes("UTF-8"));
       //Zapisivanje šifrata u fajl
       Path lokacija = Paths.get("sifrat.X509");
       Files.write(lokacija, sifrat, StandardOpenOption.CREATE_NEW);
       System.out.println("Šifrovanje je uspešno,"
               + " šifrat je upisan u fajl sifrat.X509");
       System.out.println("Šifrat (hex): "+Hex.toHexString(sifrat));
    }
}
```

*Primer 5.14 Šifrovanje sa digitalni sertifikatom*

Programski izlaz:

```
Sertifikat je učitan u memoriju.
Šifrovanje je uspešno, šifrat je upisan u fajl sifrat.X509
Šifrat (hex): 150bc6d7c111b5931661d5e84fabb93bdbaa0913fddb9aad294cb1b9c3da1eb5df51b851
c2b15ad3bf35095d3e6fa52bf05d169cac552c361277ef27c9523fbe83ad9b4ac87f2130624e00c7bf2373
b4f0e94b7d6763d5277c919f7c9534d7777247d43d86861f274844709117310c1e513caa5d056fb835cfb7
47bdba693d80c0f7b82813aca1100336c5b438ca8a09bf4a725de75e7603dc3f08a389dc1116de55dd9ded
756baa40cda96576d3e68598d97da8cac8d429cc19d059c6c3be3fb6f7395bec2880ee01c4492a6c703bd1
```

Za šifrovanje koristili smo *RSA* sertifikat kodovan u *DER* formatu. Šifarska instanca je inicijalizovana preko preko dva argumenta. Prvi argument predstavlja režim

šifrovanja (Cipher.ENCRYPT_MODE), a drugi sertifikat koji u sebi sadrži javni ključ. Dobijeni šifrat je zapisan u fajl „sifrat.X509“.

### 5.3.4 DEŠIFROVANJE SA PRIVATNIM KLJUČEM

U primeru koji sledi, zadatak je dešifrovanje poruke koja je šifrovana sa javnim ključem u prethodnom primeru. Obzirom da se privatni ključ ne nalazi u sertifikatu, a postoji negde sigurno, jer digitalni sertifikat nije bilo moguće generisati bez privatnog ključa, potrebno je pristupiti skladištu ključeva i preko adekvatnog alijasa učitati privatni ključ. Skladište ključeva ima rešenu kontrolu pristupa, gde je bilo potrebno uneti lozinku "Singidunum" za pristup entitetima. Svi entiteti za pristup koriste lozinku koja je ista kao lozinka za pristup skladištu ključeva.

```
public class DigitalniSertifikatDesifrovanje {
    public static void main(String[] args) throws KeyStoreException, FileNotFoun
        //Učitavanje skladišta ključeva
        KeyStore skladiste = KeyStore.getInstance(KeyStore.getDefaultType());
        FileInputStream fis = new FileInputStream("skladiste.keystore");
        char[] lozinka
                = new char[]{'S', 'i', 'n', 'g', 'i', 'd', 'u', 'n', 'u', 'm'};
        skladiste.load(fis, lozinka);
        //Preko alijasa RSAsertifikat uzimamo privatni ključ
        RSAPrivateKey privatniKljuc =
                (RSAPrivateKey)skladiste.getKey("RSAsertifikat", lozinka);
        Arrays.fill(lozinka, '\u0000' ) ;
        System.out.println("Učitavanje RSA privatnog ključa!");
        //Inicijalizovanje šifre za mod dešifrovanje sa privatnim ključem
        Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding") ;
        sifra.init(Cipher.DECRYPT_MODE, privatniKljuc);
        //Učitavanje šifrata iz fajla
        Path lokacija = Paths.get("sifrat.X509");
        byte sifrat[] =  Files.readAllBytes(lokacija);
        //Dešifrovanje
        byte otvorenaPoruka[] = sifra.doFinal(sifrat);
        //Prikaz otvorene poruke
        System.out.println("Otvorena poruka je: "+new String(otvorenaPoruka));
    }
}
```

*Primer 5.15 Dešifrovanje sa privatnim ključem od odgovarajućeg sertifikata*

Programski izlaz:

```
Učitavanje RSA privatnog ključa!
Otvorena poruka je: Praktikum - Zaštita informacionih sistema
```

Za dešifrovanje smo koristili privatni ključ koji je bio smešten u skladište ključeva sa obezbeđenom kontrolom pristupa. Šifarska instanca je inicijalizovana preko preko dva argumenta. Prvi argument predstavlja režim šifrovanja (Cipher.DECRYPT_MODE), a drugi privatni ključ.

Pored navedenih standardnih procedura, napominjemo da je od izuzetne važnosti brisanje podataka iz memorijske lokacije u kojoj je bila smeštena lozinka. Ova rutina nas štiti od potencijalnih napada na memoriju računara.

### 5.3.5 JEDAN SCENARIO SA RSA I TAJNIM KLJUČEM

Kombinacija *RSA* sa tajnim ključem je jedan realan scenario tajne komunikacije između Alise i Boba. Ukratko, Alisa sa Bobovim javnim sertifikatom šifruje *AES* tajni ključ koji je prethodno generisala. Alisa je sa istim *AES* tajnim ključem šifrovala bankarski dokument. Zatim, Alisa šalje šifrat tajnog ključa i šifrat bankarskog dokumenta Bobu. Bobov zadatak je da obavi kompletan postupak inverzno i da sazna informacije iz bankarskog dokumenta.

Scenario po fazama na Alisinoj strani:

1. Alisa učitava u memoriju Bobov sertifikat;
2. Alisa generiše tajni ključ za *AES* algoritam;
3. Alisa inicijalizuje *AES* šifru za šifrovanje sa digitalnim sertifikatom (javni ključ);
4. Alisa šifruje generisani tajni ključ;
5. Alisa zapisuje šifrat *AES* ključa u fajl;
6. Alisa inicijalizuje *AES* algoritam za šifrovanje;
7. Alisa učitava dokument-izvod sa bankarskim transakcijama;
8. Alisa šifruje *AES*-om dokument i šifrat zapisuje u fajl;
9. Alisa šalje Bobu šifrat tajnog ključa i šifrat bankarskog izvoda;

Scenario po fazama na Bobovoj strani:

1. Bob učitava privatni ključ iz svog skladišta ključeva;
2. Bob učitava u memoriju šifrat tajnog ključa koji je dobio od Alise;
3. Bob inicijalizuje algoritam za dešifrovanje sa privatnim ključem;
4. Bob nakon dešifrovanja dobija tajni ključ;
5. Bob učitava šifrat izvoda bankarskih transakcija;
6. Bob inicijalizuje *AES* algoritam za dešifrovanje;
7. Bob dešifruje šifrat i dobija sve tajne informacije;

Bob može opciono da obavesti Alisu o uspešno završenom kriptografskom ciklusu. Slede dva primera, prvo primer (Primer 5.16) na Alisinoj, a zatim primer (Primer 5.17) na Bobovoj strani.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

```
public class KombinacijaRSAiTajniKljucAlisa {
    public static void main(String[] args) throws CertificateException, FileNotF
        //1-Alisa učitava u memoriju Bobov sertifikat
        CertificateFactory factory = CertificateFactory.getInstance("X.509");
        FileInputStream fis = new FileInputStream(new File("RSAsertifikat.der"));
        Certificate sertifikat = factory.generateCertificate(fis);
        System.out.println("Sertifikat je učitan u memoriju.");

        //2-Alisa generiše tajni ključ za AES algoritam
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        SecretKey kljuc = kg.generateKey();
        byte[] aesKljuc = kljuc.getEncoded();
        SecretKeySpec specKljuca = new SecretKeySpec(aesKljuc, "AES");

        //3-Alisa inicijalizuje šifru za šifrovanje sa digitalnim sertifikatom
        Cipher sifrovanjeKljuca = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        sifrovanjeKljuca.init(Cipher.ENCRYPT_MODE, sertifikat);

        //4-Alisa šifruje generisani AES ključ
        byte kljucSifrat[] = sifrovanjeKljuca.doFinal(aesKljuc);

        //5-Alisa zapisuje šifrat AES ključa u fajl
        Path lokacija = Paths.get("SifratTajniKljuc");
        Files.write(lokacija, kljucSifrat, StandardOpenOption.CREATE_NEW);

        //6-Alisa inicijalizuje AES algoritam za šifrovanje
        Cipher sifrovanjeporuke = Cipher.getInstance("AES");
        sifrovanjeporuke.init(Cipher.ENCRYPT_MODE, specKljuca);

        //7-Alisa učitava dokument-izvod sa bankarskim transakcijama
        lokacija = Paths.get("BankarskeTransakcije");
        byte otvorenaPoruka[] =  Files.readAllBytes(lokacija);

        //8-Alisa šifruje AES-om dokument i šifrat zapisuje u fajl
        byte sifratPoruka[] = sifrovanjeporuke.doFinal(otvorenaPoruka);
        lokacija = Paths.get("sifratBankarskeTransakcije");
        Files.write(lokacija, sifratPoruka, StandardOpenOption.CREATE_NEW);

        //9-Alisa šalje Bobu šifrat tajnog ključa, šifrat bankarskog izvoda
        System.out.println("Scenario na Alisinoj strani je završen!");
    }
}
```

*Primer 5.16 Scenario na Alisinoj strani* – RSA *i tajni ključ*

```
public class KombinacijaRSAiTajniKljucBob {

    public static void main(String[] args) throws KeyStoreException, FileNotFoun
        //1-Bob učitava privatni ključ iz skladišta ključeva
        KeyStore skladiste = KeyStore.getInstance(KeyStore.getDefaultType());
        FileInputStream fis = new FileInputStream("skladiste.keystore");
        char[] lozinka
                = new char[]{'S', 'i', 'n', 'g', 'i', 'd', 'u', 'n', 'u', 'm'};
        skladiste.load(fis, lozinka);
        RSAPrivateKey privatniKljuc =
                (RSAPrivateKey)skladiste.getKey("RSAsertifikat", lozinka);
        Arrays.fill(lozinka, '\u0000' ) ;

        //2-Bob učitava u memoriju šifrat tajnog ključa koji je dobio od Alise
        Path lokacija = Paths.get("SifratTajniKljuc");
        byte sifratTajniKljuc[]  =  Files.readAllBytes(lokacija);

        //3-Bob inicijalizuje algoritam za dešifrovanje sa privatnim ključem
        Cipher sifra = Cipher.getInstance("RSA/ECB/PKCS1Padding") ;
        sifra.init(Cipher.DECRYPT_MODE, privatniKljuc);

        //4-Bob dobija tajni ključ za AES algoritam
        byte aesKljuc[] = sifra.doFinal(sifratTajniKljuc);
        SecretKeySpec specKljuca = new SecretKeySpec(aesKljuc, "AES");
        System.out.println("AES ključ je dobijen preko privatnog ključa!");

        //5-Bob učitava šifrat izvoda bankarskih transakcija
        lokacija = Paths.get("sifratBankarskeTransakcije");
        byte sifratBankarskeTransakcije[]  =  Files.readAllBytes(lokacija);

        //6-Bob inicijalizuje AES algoritam za dešifrovanje
        Cipher desiforvanjePoruke = Cipher.getInstance("AES");
        desiforvanjePoruke.init(Cipher.DECRYPT_MODE, specKljuca);

        //7-Bob dešifruje šifrat i dobija sve skrivane informacije
        byte otvorenaPoruka[] = sifra.doFinal(sifratBankarskeTransakcije);
        System.out.println("Otvorena poruka: "+new String(otvorenaPoruka));

        //8-Bob može opciono da izvesti Alisu o uspešnom kriptografskom postupku
        System.out.println("Scenario na Bobovoj strani je završen!");
    }
}
```

*Primer 5.17 Scenario na Bobovoj strani* – RSA *i tajni ključ*

# 6. *NIST* STATISTIČKA BATERIJA TESTOVA

*NIST*-ov paket testova predstavlja statistički paket koji se sastoji od 15 testova koji su razvijeni za testiranje slučajnosti binarne sekvence koje proizvode hardver ili softver na bazi kriptografski slučajnih ili pseudo slučajnih brojeva. Ovi testovi se fokusiraju na različite vrste determinizma koje bi mogle da postoje u nizu. Neki testovi su podeljeni u više podtestova.

## 6.1 PREGLED STATISTIČKIH TESTOVA

U narednim poglavljima ukratko ćemo da diskutujemo posebno o svakom od testova. Slede testovi iz NIST baterije testova:

- Ispitivanje učestalosti u nizu
- Ispitivanje učestalosti u bloku
- Ispitivanje uzastopnih ponavljanja istih bitova u nizu
- Ispitivanje najdužeg uzastopnog ponavljanja jedinica u bloku
- Ispitivanje stanja binarne matrice
- Ispitivanje diskretne Furijerove transformacije
- Ispitivanje nepreklapajućih uzoraka
- Ispitivanje preklapajućih uzoraka
- Maurerov univerzalni statistički test
- Ispitivanje linearne kompleksnosti
- Serijski test
- Ispitivanje približne entropije
- Ispitivanje slučajnih zbirova
- Ispitivanje slučajne digresije
- Ispitivanje slučajne promenljive digresije

### 6.1.1 ISPITIVANJE UČESTALOSTI U NIZU

U fokusu ovog testu je ispitivanje odnosa jedinica i nula u nizu bitova. Cilj je uočavanje jednakosti pojavljivanja ove dve vrednosti. Potreban je približan broj

jedinica i nula u sekvenci. Svi podtestovi koji proizilaze iz ovog testa direktno zavise od njegove uspešnosti.

Test se poziva preko metode *Frequency(n)* gde je n dužina bitnog uzorka. Metoda koristi i dodatni parametar ε koji predstavlja niz bitova generisanog preko RNG ili PRNG koji se testira.

Primer:

Ulazni parametri:

$\varepsilon$ =1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100001011100 0

$n = 100$

Obrada:

$S_{100} = -16$

$s_{obs} = 1.6$

Izlazni parametar:

$P = 0.109599$

Zaključak:
Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.2 ISPITIVANJE UČESTALOSTI U BLOKU

Test posmatra odnos jedinica i nula u n-bitnim blokovima. Cilj predstavlja uočavanje jednakosti broja jedinica i nula u svakom n-bitnom bloku. Poziv testa se vrši preko metode *BlockFrequency(M, n)* gde su:

*M* – dužina svakog bloka

*n* – dužina bitnog uzorka

Kao i kod testa za ispitivanje učestalosti u nizu, koristi se i dodati parametar $\varepsilon$.

Primer:

Ulazni parametri:

$\varepsilon$ = 1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100001011100 0

$n$ = 100

$M$ = 10

Obrada:

$N$ = 10

$X^2$ = 7.2

Izlazni parametar:

$P$ = 0.706438

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.3 ISPITIVANJE UZASTOPNIH PONAVLJANJA ISTIH BITOVA U NIZU

Posmatrana karakteristika u ovom testu je ukupan broj uzastopnih ponavljanja jednog broja u nizu. Svrha ovog testa je da se odredi da li broj uzastopnih ponavljanja nula i jedinica odgovara očekivanog slučajnoj sekvenci.

Poziv testa se vrši preko metode *Runs(n)* gde je n dužina bitnog uzorka. Takođe postoji i dodatni parametar $\varepsilon$.

Primer:

Ulazni parametri:

$\varepsilon$ = 1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100001011100 0

$n$ = 100

$\tau$ = 0.02

Obrada:

$\pi$ = 0.42

*Vn(abs)* = 52

Izlazni parametar:

$P = 0.500798$

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.4 ISPITIVANJE NAJDUŽEG UZASTOPNOG PONAVLJANJA JEDINICA U BLOKU

U ovom testu se posmatra najduže uzastopno ponavljanje jedinica u n-bitnim blokovima. Svrha je određivanje da li se dužina najdužeg uzastopnog ponavljanja poklapa sa dužinom koja bi se očekivala u nizu slučajnih brojeva.

Poziv testa vrši se preko metode *LongestRunOfOnes(n)* gde je n dužina bitnog uzorka. Takođe postoji i dodatni parametar $\varepsilon$.

$M$ – dužina svakog bloka. Test je podešen tako da za $M$ koristi tri vrednosti: $M$ = 8, $M$ = 128 i $M$ = 104 u skladu sa dužinom parametra $n$:

| Minimum n | M |
|---|---|
| **128** | **8** |
| **6272** | **128** |
| **750,000** | **104** |

Primer:

U slučaju kada je $K$ = 3 i $M$ = 8:

Ulazni parametri:

$\varepsilon$ = 11001100000010101011011000100110011100000000000001001

00110101010001000100111101011010000000110101111100

11001110011011011000010110010

$n$ = 128

Obrada:

| Maksimalno ponavljanje podbloka | Maksimalno ponavljanje podbloka |
|---|---|
| **11001100 (2)** | 00010101 (1) |
| **01101100 (2)** | 01001100 (2) |
| **11100000 (3)** | 00000010 (1) |

| | |
|---|---|
| **01001101 (2)** | 01010001 (1) |
| **00010011 (2)** | 11010110 (2) |
| **10000000 (1)** | 11010111 (3) |
| **11001100 (2)** | 11100110 (3) |
| **11011000 (2)** | 10110010 (2) |

v0 = 4; v1 = 9; v2 = 3; v4 = 0; $x_2$ = 4.882457

Izlazni parametar:

*P* = 0.180609

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.5 ISPITVANJE STANJA BINARNE MATRICE

Svrha ovog testa je provera linearne zavisnosti između podnizova fiksne dužine iz originalnog niza.

Poziv testa vrši se preko metode *Rank(n)* gde je *n* – dužina bitnog uzorka. Koristi se i dodati parametar $\varepsilon$.

Primer:

Ulazni parametri:

$\varepsilon$ = 11001001000011111101101010100010001000010110100011

00001000110100110001001100011001100010100001011100 0

$n$ = 100

Obrada:

$N_1$ = 46

$N_0$ = 47.5

$d$ = -1.376494

Izlazni parametar:

P = 0.168669

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.6 ISPITIVANJE DISKRETNE FURIJEROVE TRANSFORMACIJE

Fokus ovog testa predstavlja najviše vrednosti diskretne Furijerove transformacije niza. Svrha ovog testa je da otkrije periodične funkcije u testiranom nizu koja bi ukazivala na odstupanje od pretpostavke o slučajnosti. Namera je da se otkrije da li je broj najviših vrednosti koje prelaze 95% značajno različit od preostalih 5%.

Poziv testa vrši se preko metode *DiscreteFourierTransform(n)* gde je *n* – dužina bitnog uzorka. Koriste se i dodatni parametri:

*M* – broj redova u matrici

*Q* – broj kolona u matrici

Primer:

Ulazni parametri:

$\varepsilon$ = prvih 1000000 binarnih vrednosti proširenje $\varepsilon$

$n$ = 100000

$M$ = Q = 32

Obrada:

$N$ = 97

$F_M$ = 23, $F_{M-1}$ = 60, $N - F_M - F_{M-1}$= 14

$\chi^2$ = 1.2619656

Izlazni parametar:

P = 0.532069

Zaključak:
Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.7 ISPITIVANJE NEPREKLAPAJUĆIH UZORAKA

Karakteristika koja se posmatra u ovom testu je učestalost pojave svih mogućih $n$-bitnih uzoraka gde ne dolazi do preklapanja u celom ispitivanom nizu. Test bi trebalo da otkrije da li je broj pojava uzoraka koji se ne preklapaju približno jednak broju koji se očekuje za niz slučajnih brojeva.

Poziv testa vrši se preko metode *NonOverlappingTemplateMatching(m, n)* gde su:

$m$ – dužina bita u šablonu

$n$ – dužina bitnog uzorka

Primer**:**

Ulazni parametri:

$\varepsilon = 2^{20}$ bitova dobijenih preko G-SHA-1 generatora

$n = 2^{20}$

$B$ = 000000001

Obrada:

$\mu$ = 255.984375

$\sigma^2$ = 247.499999

$W1$ = 259; $W2$ = 229; $W3$ = 271; $W4$ = 245; $W5$ = 272; $W6$ = 262;

$W7$ = 259; i $W8$ = 246

$\chi^2(abs)$ = 5.999377

Izlazni parametar:

$P$ = 0.110434

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.8 ISPITIVANJE PREKLAPAJUĆIH UZORAKA

Karakteristika koja se posmatra u ovom testu je učestalost pojave svih mogućih n-bitnih uzoraka gde dolazi do preklapanja u celom ispitivanom nizu. Test bi trebalo da otkrije da li je broj pojava uzoraka koji se preklapaju približno jednak broju koji se očekuje za niz slučajnih brojeva.

Poziv testa vrši se preko metode *OverlappingTemplateMatching(m, n)* gde su

*m* – dužina bita u šablonu

*n* – dužina bitnog uzorka

Primer:

Ulazni parametric:

$\varepsilon$ = binarno proširenje $\varepsilon$ do 1000000 bita

*n* = 1000000

*B* = 111111111

Obrada:

*v0* = 329; *v1* = 164; *v2* = 150; *v3* = 111; *v4* = 78; i *v5* = 136

$\chi 2$ (abs) = 8.965859

Izlazni parametar:

*P* = 0.110434

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.9 MAUREROV UNIVERZALNI STATISTIČKI TEST

Ovim testom se ispituje mogućnosti kompresije niza bez gubitka informacije. Za niz koji se može značajno kompresovati smatra se da nije niz slučajnih bitova zbog činjenice da je kompresija niza efikasnija ukoliko niz pokazuje periodična svojstva.

Poziv testa vrši se preko metode *Universal(L, Q, n)* gde su

*L* – dužina svakog bloka

*Q* – broj blokova u inicijalnoj sekvenci

*n* – dužina bitnog uzorka

Takođe postoji i dodatni parametar *ε*.

Primer:

Ulazni parametri:

*ε* = binarno proširenje ε do 1000000 bita

*n* = 1000000

*B* = 111111111

Obrada:

*v0* = 329; *v1* = 164; *v2* = 150; *v3* = 111; *v4* = 78; i *v5* = 136

$\chi^2$ (abs) = 8.965859

Izlazni parametar:

*P* = 0.110434

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.10 ISPITIVANJE LINEARNE KOMPLEKSNOSTI

Ovaj test se koristi za dobijanje dužine linearnog pomeračkog registra (engl. *Linear feedback shift register* - LFSR). Svrha ovog testa je utvrđivanje da li je sekvenca dovoljno kompleksa da se može smatrati slučajnom.

Poziv testa vrši se preko metode *LinearComplexity(M, n)* gde su:

*M* – dužina svakog bloka

*n* – dužina bitnog uzorka

Koristi se i dodati parametar $\varepsilon$ kao i *K* – broj stepeni slobode.

Primer:

Ulazni parametri:

$\varepsilon$ = prvih 1000000 binarnih vrednosti proširenje $\varepsilon$

*n* = 1000000 = 106

*M* = 1000

Obrada:

*v0* = 11; *v1* = 31; *v2* = 116; *v3* = 501; *v4* = 258; *v5* = 57; *v6* = 26

$\chi 2$(abs) = 2.700348

Izlazni parametar:

*P* = 0.845406

Zaključak:

Obzirom da je $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.11 SERIJSKI TEST UČESTALOSTI

Test se koristi za utvrđivanje učestalosti svih mogućih preklapanja *n*-bitnog niza u čitavoj sekvenci.

Poziv testa vrši se preko metode *Serial(m, n)* gde su:

*m* – dužina svakog bloka

*n* – dužina bitnog uzorka

Koristi se i dodati parametar $\varepsilon$.

Primer**:**

Ulazni parametri:

$\varepsilon$ = 1000000 bita binarnog proširenje ε

$m$ = 2

$n = 1000000 = 10^6$

Obrada:

#0s = 499971

#1s = 500029

#00s = 250116

#01s = #10s = 249855

#11s = 250174

$\psi^2_2 = 0.343128$; $\psi^2_1 = 0.003364$; $\psi^2_0 = 0.000000$

Izlazni parametar:

$\nabla\psi^2_2 = 0.339764$

$\nabla^2\psi^2_2 = 0.336400$

$P_1 = 0.843764$

$P_2 = 0.561915$

Zaključak:

Obzirom da su $P_1$ i $P_2 \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.12 ISPITIVANJE PRIBLIŽNE ENTROPIJE

U ovom testu posmatra se takođe učestalost pojave svih mogućih preklapajući *n*-bitnih uzoraka u nizu. Cilj je poređenje učestalosti preklapajućih blokova sa očekivanim rezultatima.

Poziv testa vrši se preko metode *ApproximateEntropy(m, n)* gde su:

*m* – dužina svakog bloka

*n* – dužina bitnog uzorka

Koristi se i dodati parametar $\varepsilon$.

Primer:

Ulazni parametri:

$\varepsilon$ = 1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100001011100 0

*m* = 2 i *n* = 100

Obrada:

*ApEn(m)* = 0.665393

$\chi^2$(obs) = 5.550792

Izlazni parametar:

*P* = 0.235301

Zaključak:
Obzirom da jesu $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.13 ISPITIVANJE SLUČAJNIH ZBIROVA

Svrha ovog testa je potreba da se utvrdi da li je kumulativni zbir parcijalnih sekvenci koje se javljaju u nizu koji se testira suviše veliki ili suviše mali u odnosu na očekivano ponašanje tog kumulativnog zbira slučajnih sekvenci.

Poziv testa vrši se preko metode *CumulativeSums(mod, n)* gde je:

mod = 0 ili mod = 1

n – dužina bitnog uzorka

Koristi se i dodati parametar $\varepsilon$.

Primer:

Ulazni parametri:

ε = 11001001000011111101101010100010001000010110100011

00001000110100110001001100011001100010100010111000

n = 100

mod = 0 || mod = 1

Obrada:

z = 1.6 || z = 1.9

Izlazni parametar:

*P* = 0.219194 || *P* = 0.114866

Zaključak:

Obzirom da jesu $P \geq 0.01$, smatra se da je sekvenca slučajna.

### 6.1.14 ISPITIVANJE SLUČAJNE DIGRESIJE

Svrha ovog testa je da se utvrdi da li je broj određenih stanja u okviru ciklusa odstupa od onoga što bi se očekivalo od slučajne sekvence. Ovaj test je zapravo niz od osam testova (i zaključaka), jedan test i zaključak za svako od stanja: -4, -3, -2, -1 i +1, +2, +3, +4.

Poziv testa vrši se preko metode *RandomExcursions(n)* gde je *n* – dužina bitnog uzorka. Koristi se i dodati parametar ε.

Primer:

Ulazni parametri:

*ε* = 1000000 bita binarnog proširenje ε

*n* = 1000000 = 106

Obrada:

*J* = 1490

| Stanje = x | $x^2$ | P | Zaključak |
|---|---|---|---|
| **-4** | 3.835698 | 0.573306 | slučajan |
| **-3** | 7.318707 | 0.197996 | slučajan |
| **-2** | 7.861927 | 0.164011 | slučajan |
| **-1** | 15.692617 | 0.007779 | nije slučajan |

| | | | |
|---|---|---|---|
| **+1** | 2.485906 | 0.778616 | slučajan |
| **+2** | 5.429381 | 0.365752 | slučajan |
| **+3** | 2.404171 | 0.790853 | slučajan |
| **+4** | 2.393928 | 0.792378 | slučajan |

Zaključak:

Za *x*, u sedam stanja važi da je $P \geq 0.01$, što dovodi do zaključka da je sekvenca slučajna.

### 6.1.15 ISPITIVANJE SLUČAJNE PROMENLJVE DIGRESIJE

Svrha testa je da se utvrdi da li je ukupan broj stanja u kojima se niz nalazi, između [-9, -1] i [1, 9] premašuje očekivane vrednosti čisto slučajne sekvence. Ovaj test zapravo predstavlja grupu od 18 testova (i zaključaka), svaki test i zaključak za svako stanje: -9, -8,..., -1i +1, +2,...,+9.

Poziv testa vrši se preko metode *RandomExcursionsVariant(n)* gde je *n* – dužina bitnog uzorka. Koristi se i dodati parametar ε.

Primer:

Ulazni parametri:

$\varepsilon$ = 1000000 bita binarnog proširenje $\varepsilon$

*n* = 1000000 = 106

Obrada:

$J$ = 1490

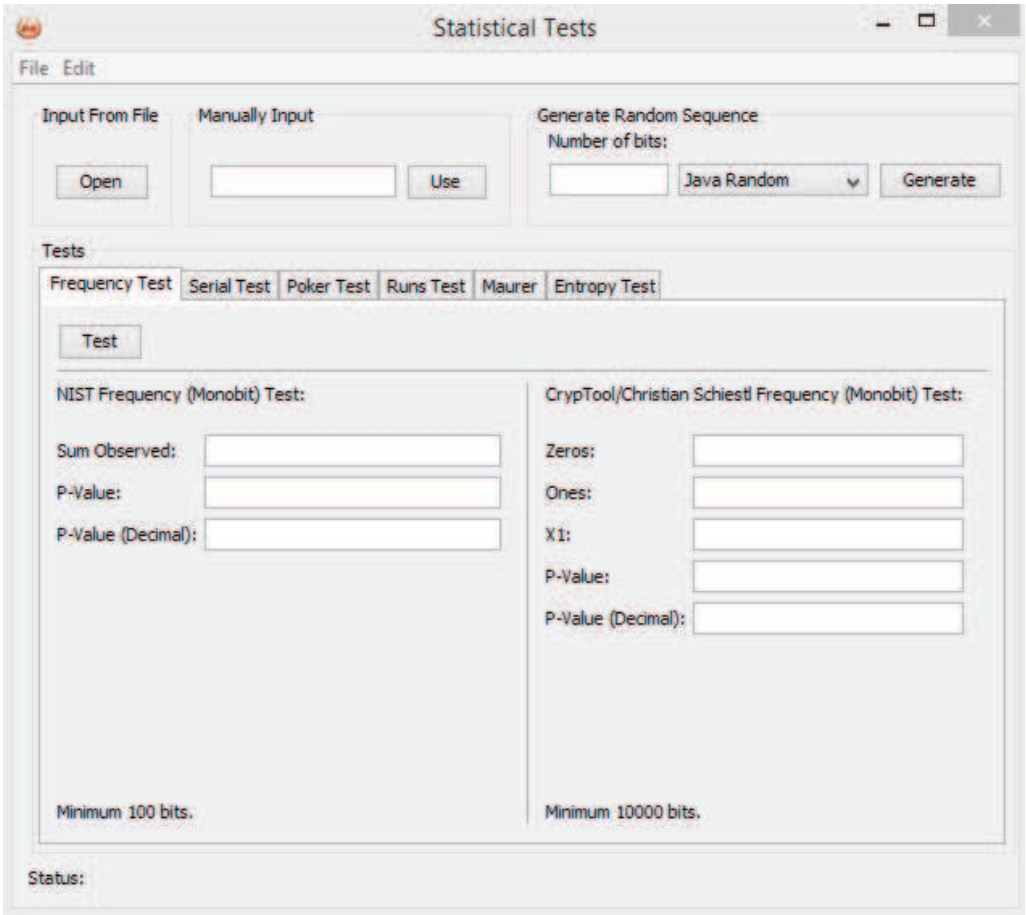
| Stanje = $x$ | $x^2$ | $P$ | Zaključak |
|---|---|---|---|
| **-4** | 3.835698 | 0.573306 | slučajan |
| **-3** | 7.318707 | 0.197996 | slučajan |
| **-2** | 7.861927 | 0.164011 | slučajan |
| **-1** | 15.692617 | 0.007779 | nije slučajan |
| **+1** | 2.485906 | 0.778616 | slučajan |
| **+2** | 5.429381 | 0.365752 | slučajan |
| **+3** | 2.404171 | 0.790853 | slučajan |
| **+4** | 2.393928 | 0.792378 | slučajan |

Zaključak:

Obzirom da je $P \geq 0.01$ za svako od 18 stanja za $x$, smatra se da je sekvenca slučajna.

## 6.2 JAVA APLIKACIJA SA IMPLEMENTIRANOM BATERIJOM TESTOVA

Za potrebe predmeta - Zaštita u informacionim sistemima – razvijena je Java aplikacija sa grafičkim interfejsom, koja implementira *NIST* i *Cryptool* statističke testove, koji predstavljaju elementarne testove kriptografske baterije testova.




*Slika 6.1 Aplikacija sa baterijom statističkih testova*

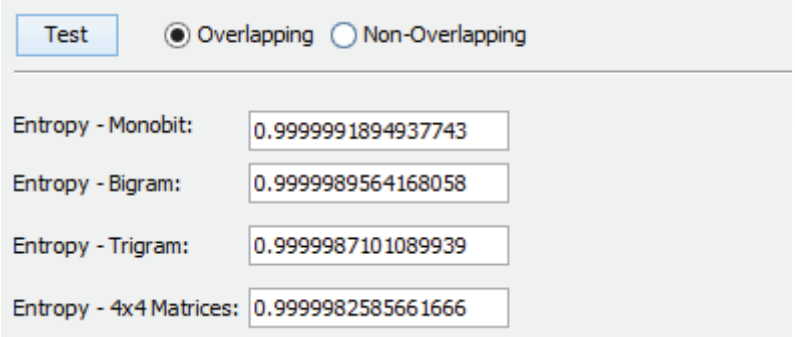
Na slici je prikazan izgled aplikacije. Po tabovima aplikacije su smešteni sledeći statistički testovi sa implementacijom po *NIST* i *Cryptool* standardu:

- Frekventni test;

- Serijski test;
- Poker test;
- *Runs* test;
- Maurerov test;
- Entropijski test;

Prikazana aplikacija nam omogućava da učitamo podatke iz fajla u zahtevanom formatu, posle čega možemo da primenimo sve dostupne testove ili da radi demonstracije testova generišemo slučajne podatke preko pseudo slučajnog algoritma u Javi. Koristićemo `SecureRandom` klasu, iz razloga što Java upravo ovu klasu koristi za generisanje ključeva. Kod nekih testova postoji ograničenje u vidu minimalnog uzorka koji je potreban da bi se podaci izložili određenom testu. Za potrebe entropijskog testa generisali smo 1.000.000 bitova.

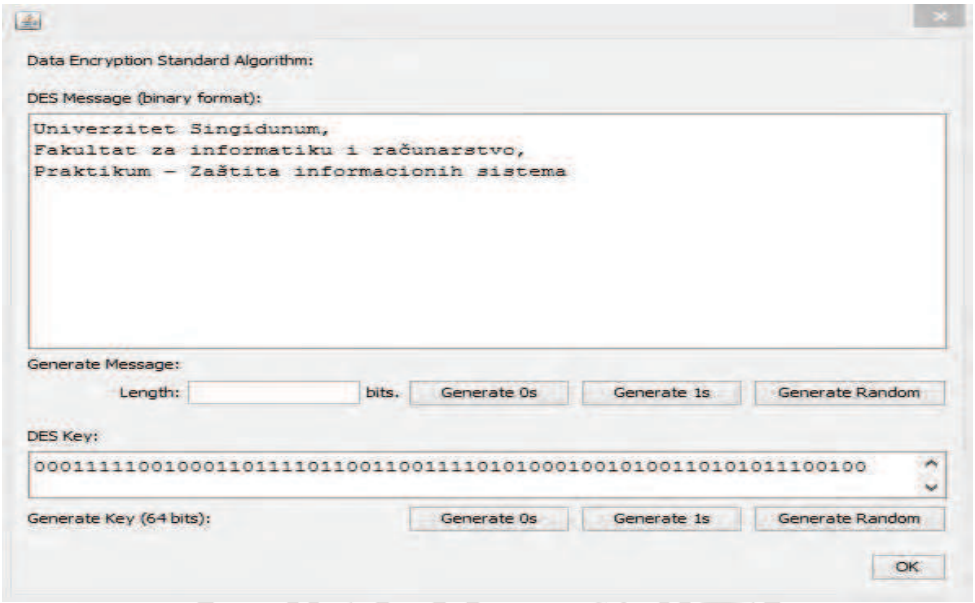
Na sledećem primeru (Primer 6.1) pokazan je rezultat entropijskog testa. Testirani podaci su prošli monobitnu, bigramsku, trigramsku i matričnu (4x4) entropijsku analizu sa preklapanjem.




*Primer 6.1 Rezultati entropijskog testa sa preklapanjem*

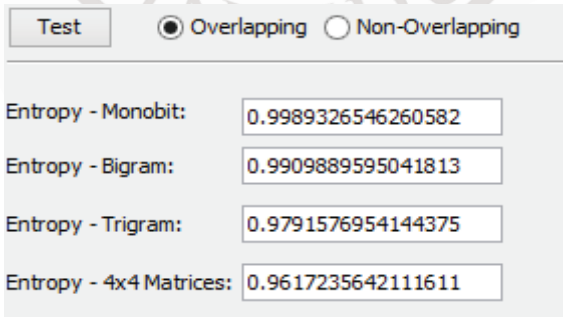
Na rezultatima vidimo da su postignute maksimalne entropijske vrednosti po jednom bitu. Količina informacije u 1 bitu je približno 1. Aplikacija pruža mogućnost izvoza testiranog binarnog niza. Pored generisanja slučajnih bitova, data je mogućnost generisanja slučajnih vrednosti preko *DES* simetričnog algoritma. Osnovna ideja je da se šifrat *DES* algoritma koristi kao izvor slučajnih podataka. Zbog visokog stepena nelinearnosti koje proizvode S-kutije u *DES* algoritmu, ovakva upotreba *DES* algoritma

je moguća. Na sledećoj slici prikazan je prozor aplikacije za podešavanje *DES* algoritma za generisanje slučajnih sekvenci. Uneli smo prvo neku slučajno smišljenu poruku i ključ u binarnom formatu koji koristimo za šifrovanje.




*Slika 6.2 Generisanje slučajnih nizova preko DES šifrat*

U primeru (Primer 6.2), prikazani su rezultati entropijskog testa za *DES* šifrat:




*Primer 6.2 Entropijski test – DES šifrat*

Rezultati ovog entropijskog testa, zanemarljivo su lošiji od testa binarnih nizova generisanih preko `SecureRandom` klase.

# 7. JEDAN PRIMER SINTEZE SOPSTVENOG IZVORA SLUČAJNOSTI

U ovom delu udžbenika diskutovaćemo o jednom potencijalnom izvoru informacija. Uloga ovakvih izvora je da se izgradi pouzdan generator kriptoloških ključeva. Pored teorijske osnove o kojoj će bit više reči, demonstriraćemo jedan primer Java koda koji u realnom vremenu prikuplja podatke od civilnog avio saobraćaja. Na osnovu ovog izvora moguća je sinteza generatora slučajnih vrednosti.

Ovako dobijene podatke potrebno je dalje analizirati i obraditi nekom od tehnika iz oblasti digitalne obrade signala, u cilju dobijanja čisto slučajnih binarnih sekvenci. Njihova svrha prvenstveno treba da bude zamena za deterministične generatore koji se koriste u Java *JCA* arhitekturi (pseudo slučajne generatore).

Zapravo, ovaj izvor daje studentima mogućnost da u svojim kriptografskim programima koriste sopstvene izvore slučajnosti za potrebe inicijalizovanja kriptografskih algoritama za generisanje simetričnih ili asimetričnih ključeva (`SecureRandom` i `SecretKey`).

Pored razvoja sopstvenog izvora slučajnosti, studenti imaju mogućnost da u svojim praktičnim radovima dizajniraju sisteme za distribuciju kriptoloških ključeva preko javnih komunikacionih kanala. Na ovaj način ukazujemo na potrebu i atraktivnosti tema koje se odnose na potpuno novo polje u kriptografiji, a to je bezbednost na fizičkom sloju.

## 7.1 ULOGA IZVORA SLUČAJNOSTI U KRIPTOGRAFIJI

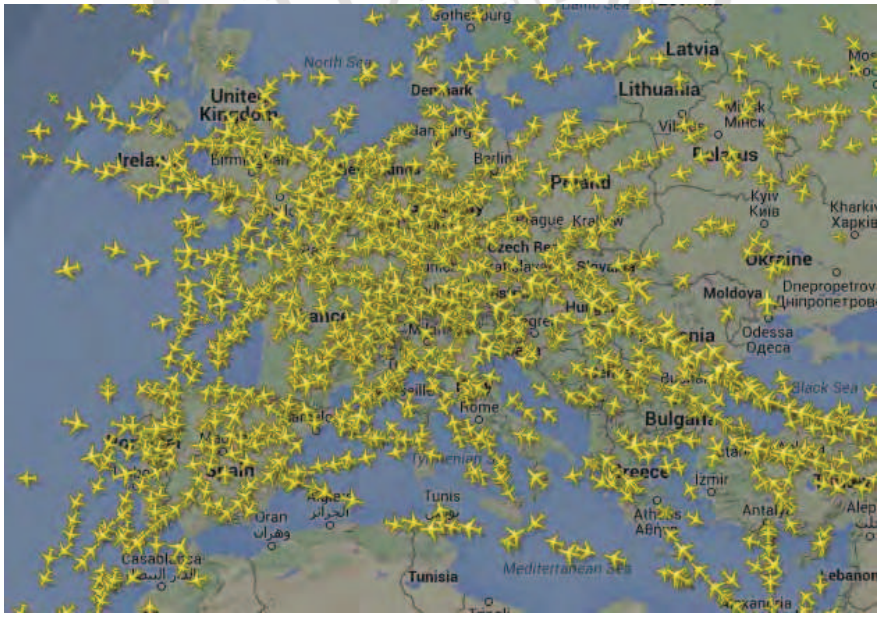
Na početku udžbenika rekli smo da je jedini teorijski dokaziv šifarski sistem upravo *One-Time* pad. Da bi ovaj šifarski sistem obezbedio perfektnu tajnost neophodno je u potpunosti ispuniti sledeće uslove: ključevi treba da budu generisani na potpuno slučajan način (ovakve izvore je moguće pronaći u prirodi), dužina ključa mora da bude jednaka dužini poruke koja se šifruje i ključ nikada ne sme da se koristi dva puta.

Naš prvi zadatak je generisanje slučajnih nizova. Slučajne nizove je moguće generisati pomoću TRNG-a i PRNG-a. TRNG predstavlja generator čisto slučajnih nizova koji je moguće konstruisati u prirodnim okruženjima, dok PRNG generiše pseudo slučajne nizove na osnovu unutrašnjih stanja generatora koja poseduju osobine čisto slučajnih nizova. Obe vrste generatora je moguće upotrebiti za projektovanje šifarskih sistema, ali u oba slučaja je neophodno generisati čisto slučajne nizove, čija je jedina uloga u domenu generisanja kriptoloških ključeva.

Uloga ovakvih izvora slučajnosti je u obezbeđivanju kriptoloških ključeva visokog kvaliteta, dovoljnih dužina i stvaranju uslova za sintezu sistema za distribuciju kriptoloških ključeva.

## 7.2 GENERISANJE SLUČAJNOSTI NA OSNOVU AVIO SAOBRAĆAJA

U svakom trenutku iznad nas nalazi se nekoliko hiljada putničkih i teretnih aviona. Ako pogledamo samo Evropu, broj letova u manje frekventnom delu dana je približno 1000, dok u nekim delovima dana taj broj se može značajno uvećati.



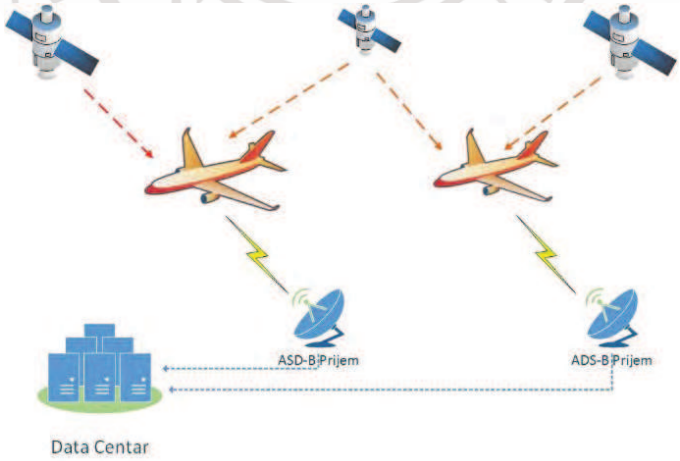

*Slika 7.1* FlightRadar24 *mapa– Evropa*

U današnje vreme, skoro svi letovi se prate preko *GPS* sistema za navigaciju. Obzirom na to da su podaci civilnog vazduhoplovstva javno dostupni, bilo gde da se nalazimo na planeti zemlji, možemo sa velikom preciznošću da znamo geografsku širinu, geografsku dužinu, brzinu i nadmorski visinu letilice. Zbog velike brzine letelica i preciznosti *GPS* sistema, koordinate se veoma brzo menjaju u jedinici vremena, a mogućnost da se letelica u istom vremenskom trenutku nađe na istoj geoinformacionoj lokaciji ni teorijski ni praktično ne postoji.

Međutim, ukoliko posedujemo informacije o više letova u istom vremenskom trenutku, odbacujemo i mogućnost da vrednosti geoinformacione lokacije svih tih letova budu u jedinici vremena predvidljive. Na osnovu ove pretpostavke, analizirali smo podatke letova na različitim geoinformacionim lokacijama u jednom vremenskom trenutku.

Prednost ovog sistema je što omogućava pristup istim podacima, na dva računara koja su fizički udaljena. Ova činjenica potencijalno obezbeđuje realizaciju sistema za distribuciju (razmenu) kriptoloških ključeva između više korisnika, pored mogućnosti za razvoj izvora slučajnosti za generisanje kriptoloških ključeva.

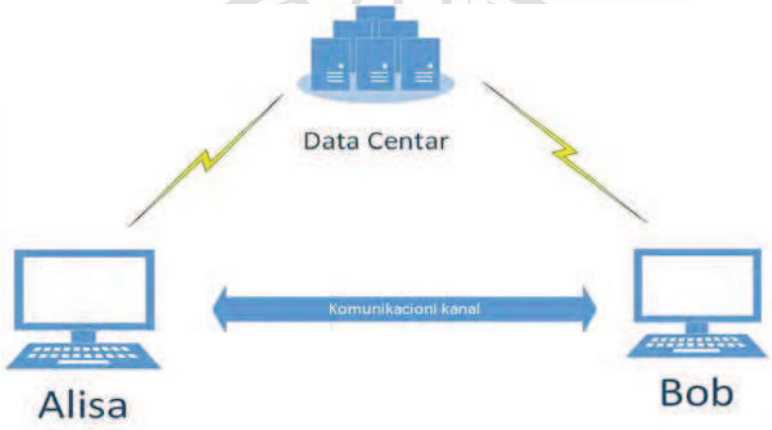
### 7.2.1 ARHITEKTURA IZVORA SLUČAJNOSTI




*Slika 7.2* ADS-B

Kao sto je prikazano na slici (Slika 7.2), *GPS* sateliti komuniciraju sa uređajima u avionu i na taj način obezbeđuju podatke potrebne za upravljanje avionom. U sledećem koraku avioni preuzimaju ulogu satelita i sada oni emituju novi radio signal u okviru koga šalju podatke relevantne za taj let. Ti podaci se prikupljaju u centrima za kontrolu letenja na zemlji, ali te podatke prema zakonima civilnog vazduhoplovstva mogu prikupljati svi zainteresovani pomoću *ADS-B* tehnologije.

Obzirom na to da se podaci na ovaj način ne mogu prikupiti samo na jednoj lokaciji za celu Evropu, iz tog razloga podaci se prikupljaju na više lokacija i smeštaju u baze podataka na Internetu, kako bi bili korišćeni za servise praćenja aviona u realnom vremenu. Jedan takav servis koji ćemo iskoristiti, prikazan je na slici (Slika 7.3).



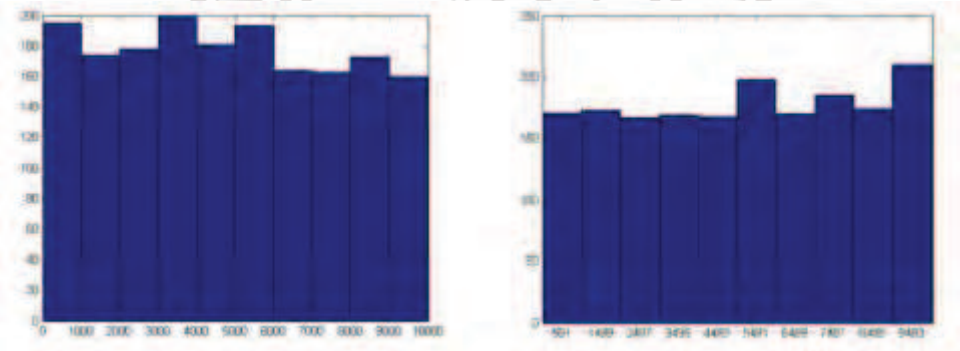

*Slika 7.3 Komunikacija između korisnika i baze podataka*

## 7.2.2 PRIKUPLJANJE I OBRADA PODATAKA

Posle prikupljenih podataka, sledi faza filtriranja. Faza filtriranja podrazumeva odbacivanje podataka o letilicama koje su još u fazi sletanja ili poletanja. Druga faza podrazumeva još i selekciju podataka. Obzirom na to da se u bazi osim podataka o geografskoj širini i dužini, nadmorskoj visini i brzini aviona nalaze i podaci o broju leta, polaznom i dolaznom aerodromu kao i podaci o avio kompaniji, izdvajamo samo podatke čije se vrednosti menjaju u jedinici vremena.

Izvršenom prostom analizom utvrdili smo da su nama od značaja samo promenljive koje se odnose na podatke o geografskoj širini i dužini, dok brzina i visina imaju kontinuitet. Dolazimo i do poslednjeg procesa u fazi filtriranja. Podaci o geografskoj širini i dužini predstavljeni su sa realnim dvocifrenim brojem sa četiri decimale. Usled velike preciznosti u decimalnom zapisu, brojevi ispred decimalne tačke nemaju trenutnu promenu, iz tog razloga se odbacuju, a zadržavaju se samo decimalne vrednosti posle decimalnog zareza. Na primer, geografska dužina je 50.4568, a geografska širina 14.2563, tada zadržavamo vrednosti posle decimalne tačke, vrednost 4568 i 2563.

Analizom čiji je rezultat prikazan na slici levo (Slika 7.4), jasno se vidi da usled zaokurživanja decimalne vrednosti u intervalima od 0000 do 0500 i od 9500 do 9999 su dosta učestalije u odnosu na vrednosti iz drugih intervala, iz tog razloga smo odstranili ova dva opsega slika desno (Slika 7.4). Ovu anomaliju možemo drugačije da interpretiramo, kao grešku vidljivu u našem sistemu usled zaokurživanja realnih brojeva na četiri decimale.



*Slika 7.4 Frekvencije po intervalima*

Nakon faze kodovanja sledi provera kvaliteta generisanih binarnih sekvenci.

### 7.2.3 ANALIZA KVALITETA IZVORA SLUČAJNOSTI

Kako bi potvrdili polaznu pretpostavku da nizovi bitova generisani na ovaj način imaju visok stepen slučajnost, koristili smo propisane statističke testove Nacionalnog

instituta za standarde i tehnologije SAD-a (NIST) i to objavljene i definisane u radu 800-22 iz Aprila 2010 pod nazivom "*A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications*".

Ovi testovi su razvijeni za testiranje slučajnosti binarne sekvence na taj način što se fokusiraju na različite vrste neslučajnosti koje bi mogle da postoje u nizu. Paket testova se sastoji od 15 različitih statističkih testova gde neki od testova su podeljeni u podtestove.

Za naše testiranje izabrali smo najčešće korišćene testove čiji rezultati se mogu smatrati visoko pouzdanim. Kako bi dobili uporedni test sa nekim već proverenim *TRN* generatorom koristili smo podatke sa sajta random.org[LINK] koji važe za potpuno slučajne podatke.

**Test 1: Ispitivanje učestanosti u nizu**

Cilj ovog testa je ispitivanje odnosa jedinica i nula u nizu bitova. Potreban je približan broj jedinica i nula u sekvenci.

*Tabela 7.1 Rezultati testa učestanosti*

| | **random.org** | **Avio izvor** |
|---|---|---|
| **P-vrednost** | 0.808976294576234 | 0.8146073609727413 |

Broj bita na kojima je izvršen kako ovaj, tako i naredni testovi je 9856. Nakon izvršenog testa možemo zaključiti da su oba niza zadovoljila uslov postavljen od strane *NIST*-a

$$P \geq 0{,}01$$

i da se mogu smatrati slučajnim. Ako poredimo rezultate možemo zaključiti da je odnos nula i jedinica u oba niza bio približno jednak.

**Test 2: Serijski test**

Test se koristi za utvrđivanje učestanosti svih mogućih preklapanja n-bitnog niza u čitavoj sekvenci.

*Tabela 7.2 Rezultati serijskog testa*

| | **random.org** | **Avio izvor** |
|---|---|---|
| **Trigrami** | | |
| **000** | 1215 | 1208 |
| **001** | 1250 | 1209 |
| **010** | 1280 | 1157 |
| **011** | 1195 | 1225 |
| **100** | 1251 | 1209 |
| **101** | 1225 | 1173 |
| **110** | 1195 | 1125 |
| **111** | 1245 | 1215 |
| **P-val 1** | 0.3174364812591314 | 0.6440496128821558 |

| | | |
|---|---|---|
| **P-vrednost** | 0.1088774362443912 | 0.4608606318159197 |

Rezultati dobijeni u ovom testu su ispunili *NIST*-ov zahtev definisan u formuli (1) te i nakon ovog testa možemo tvrditi da su nizovi slučajni, uz napomenu da je test nad našim nizom pokaza da je odnos u preklapanju ujednačeniji.

**Test 3: Ispitivanje uzastopnih ponavljanja**

Posmatrana karakteristika u ovom testu je ukupan broj uzastopnih ponavljanja jedinice ili nule u nizu.

*Tabela 7.3 Rezultati testa uzastopnih ponavljanja*

| | **random.org** | **Avio izvor** |
|---|---|---|
| **P-vrednost** | 0.6426910024891495 | 0.353825079052692 |

Nakon sprovedenog testa rezultati su zadovoljili definisane zahteve i na osnovu toga generisane nizove smatramo slučajnim. U poređenju ova dva niza nakon testa možemo zaključiti da niz sa sajta random.org ima više promena iz 0 u 1 ili obrnuto.

**Test 4: Entropijski test**

U ovom testu posmatra se učestalost pojave svih mogućih preklapajućih n-bitovskih uzoraka u nizu.

*Tabela 7.4 Rezultati entropijskog testa*

| | **Random.org** | **Avio izvor** |
|---|---|---|
| **Monobit** | 0.9999957227359038 | 0.9999958775044104 |

| | | |
|---|---|---|
| **Bigram** | 0.9999878164470851 | 0.9999625655234716 |
| **Trigram** | 0.9998791930081991 | 0.9999120107423237 |
| **4x4 Matrice** | 0.9998558732137256 | 0.9997868840346474 |

Vrednosti i ovog testa zadovoljavaju zahteve *NIST*-a. Možemo tvrditi da je naš niz slučajan i da je u poređenju sa nizom sa sajta random.org i nakon ovog testa pokazao bolje rezultate te shodno tome možemo tvrditi da poseduje osobine slučajnosti i da može biti korišćen kao materijal za generisanje kvalitetnih kriptoloških ključeva.

Zbog brzine Interneta mi nismo u mogućnosti da pratimo kretanje određene letilice za svakih 15ms, tako da nam to i dodatno pomaže da se vrednosti koordinata drastično promene. Na ovaj način u vremenskom intervalu od nekoliko sekundi možemo dobiti veliki broj binarnih nizova. Međutim broj bitova u sekundi koji proizvede neki *TRNG* je daleko veći.

### 7.2.4 JAVA PROGRAM ZA PRIKUPLJANJE PODATAKA U REALNOM VREMENU

U primeru Java koda koji sledi, prikupljamo podatke pojedinačno o svakom letu. Adresa preko koje pristupamo odnosi se na pristup podacima generisanim nad Evropom. Promenom adrese moguće je pribavljati i podatke sa drugih kontinenata. Program ispisuje dva tipa rezultata. U slučaju A, ispisujemo samo koordinate, a u slučaju B koordinate i dodatne podatke

```
public class KoridinateLetova {

    public static void main(String[] args) throws MalformedURLException, IOExcep
        String adresa = "http://db8.flightradar24.com/zones/europe_all.js?callba
        URL url = new URL(adresa);
        URLConnection con = url.openConnection();
        InputStream is = con.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String line = br.readLine();
        String SplitArgument1 = "[\\[\\]]+";
        String[] Tokens = line.split(SplitArgument1);
        String SplitArgument2 = "[,]";
        for (int i = 1; i < Tokens.length; i = i + 2) {
            System.out.println(Tokens[i]); //Iispisuje dodatne podatke o avionu
            //ISPIS SAMO KOORDINATA
            String[] Print = Tokens[i].split(SplitArgument2);
            //System.out.println(Print[1] + "  " + Print[2]);
        }
    }
}
```

*Primer 7.1 Prikupljanje podataka o trenutnim letovima*

Izlaz iz programa (slučaj A):

```
40.5271  22.9720
50.0999  14.2559
50.1156  14.2681
50.0864  14.1998
40.9728  28.8146
50.0867  14.2704
43.6153  1.3642
40.9019  29.3081
49.0514  3.0410
39.4770  28.2217
51.3102  8.2339
60.5871  37.4344
51.4493  12.8297
51.4729  -0.4705
```

Izlaz iz programa (slučaj B):

```
"ABD30F",51.1797,8.0075,227,13850,295,"3105","F-EDLE2","B77L","N861FD",1421403724,"
CAN","","FX5030",0,-1536,"FDX5030",0
"896195",60.1578,37.6818,164,37000,528,"0444","F-UUBI1","A388","A6-EEV",1421403725,
"SFO","DXB","EK226",0,0,"UAE226",0
"3C4587",51.4328,12.6258,261,4850,178,"1000","F-EDDC2","B77L","D-AALG",1421403726,"
PVG","","3S571",0,0,"BOX571",0
"7C4927",51.4729,-0.4705,61,0,0,"4744","T-EGLL21","A388","VH-OQH",1421403718,"DXB",
"LHR","QF1",1,0,"QFA1",0
"89618E",60.8592,37.2748,163,37000,521,"3761","F-ULPB4","A388","A6-EEO",1421403727,
"LAX","DXB","EK216",0,-128,"UAE216",0
"396441",49.0048,2.7921,265,3000,170,"4303","N-LFPM1","A332","F-GZCB",1421403725,"G
IG","CDG","AF445",0,-896,"AFR445",0
```

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

# DODATAK - PROJEKTNI ZADACI I LABORATORIJSKE VEŽBE

Naš osnovni cilj u ovom udžbeniku je da student izborom predmeta „Praktikum – zaštita informacionih sistema“ u potpunosti praktično ovlada izabranom oblašću. Studenti prolaze kroz nekoliko faza u toku trajanja ovog predmeta:

1. Izbor teme iz oblasti zaštite u informacionim sistemima;
2. Istraživanje aktuelnog stanja iz izabrane oblasti i usvajanje teorijskih znanja;
3. Izrada praktičnog rešenja za demonstraciju bezbednosnih problema i rešenja za aktuelne probleme;
4. Predstavljanje rezultata istraživanja - pregled aktuelnog stanja u izabranoj oblasti;
5. Predstavljanje praktičnih rezultata - demonstracija praktičnog rešenja;
6. Izrada projektnog/diplomskog rada na zadatu temu;

## 8.1 TEME ZA PROJEKTNI ZADATAK

Sledeće teme su potencijalni projektni zadaci. Takođe, studenti mogu predložiti temu koja može uz odobrenje da postane tema za projektni / diplomski rad:

1. Jedan primer pravilne implementacija kriptografskih mehanizama u Java programskom jeziku;
2. Razvoj kriptografskog protokola u Java programskom jeziku;
3. Implementacija kriptografskih algoritama u poslovne aplikacije (kriptografska zaštita fajlova i baza podataka);
4. Algoritmi za sekvencijalne ili blokovske šifre i njihove modifikacije (rad na performansama algoritma);
5. Standard za autentifikaciju i digitalni identitet korisnika na Internetu, koji omogućava korisnicima da se jednim isti nalogom autentifikuju na različitim *Veb* sajtovima (*OPEN-ID*);
6. Zaštita izvornog koda *PHP* aplikacija;
7. Kriptografske funkcije u *Veb* okruženju;
8. Upravljanje bezbednošću *Veb* aplikacija sa otvorenim izvornim kodom;
9. Dizajniranje novih režima za šifrovanje kod blokovskih šifara (*CBC*, *ECB*, *CFB*);

10. Razvoj i implementacija *CAPTCHA* sistema;
11. *SQL injection* napadi na *Veb* aplikacije;
12. *Cross-Site Scripting* (*XSS*) i *Cross-site request forgery (CSRF);*
13. Zaštita *Veb* aplikacija i baza u deljenom produkcionom okruženju;
14. Sistemi za elektronsku trgovinu – bezbednosni i funkcionalni aspekt;
15. Biometrijski sistemi za autentifikaciju i informaciona analiza biometrije;
16. Implementacija biometrijske autentifikacije;
17. Generisanje kriptoloških ključeva na osnovu biometrije;
18. Distribucija kriptoloških ključeva preko javnih komunikacionih kanala (bezbednost na fizičkom sloju);
19. Razvoj servisa za digitalno potpisivanje dokumenata sa sopstvenim sertifikacionim telom;
20. Primena informacionih mera u kriptoanalizi;
21. Primena informacionih mera u procesu estimacije čisto slučajnih nizova;
22. Razvoj sopstvene *PKI* infrastrukture u *LAN*-u;
23. Konstruisanje pseudo slučajnog generatora (*PRNG*) na osnovu slučajnih početnih stanja dobijenih preko nekog fizičkog izvora informacija (*TRNG*);
24. Analiza steganografskih tehnika i primena u oblasti digitalnih vodenih pečata u cilju zaštite autorskih prava;
25. Razvoj i primena pesudo perfektnih šifara u današnjem okruženju sa približnim bezbednosnim performansama *One-time pad* šifre;
26. Razvoj sopstvenih kriptografskih protokola za primenu u integrisanim sistemima;
27. Uloga distribuiranog računarstva u razvoju kriptoanalize;
28. Razvoj i primena savremenih kriptoanalitičkih metoda;
29. Standardi za implementaciju i testiranje kriptografskih algoritama;
30. Razvoj ergonomskih aplikacija za kriptografsku zaštitu fajlova na operativnom sistemu;
31. Merenje performansi simetričnih šifri i optimizacija za primenu na određenoj platformi;
32. Modul za generisanje kriptoloških ključeva na osnovu sopstvenih izvora slučajnosti;
33. Kriptografska zaštita u bazama podataka;
34. Protokoli za razmenu simetričnih ključeva;

35. Aplikacija za digitalno potpisivanje dokumenata sa modulom za generisanje i skladištenje sertifikata;
36. Analiza performansi provajdera kriptografskih funkcija na Java bezbednosnoj arhitekturi;

## 8.2 LABORATORIJSKE VEŽBE

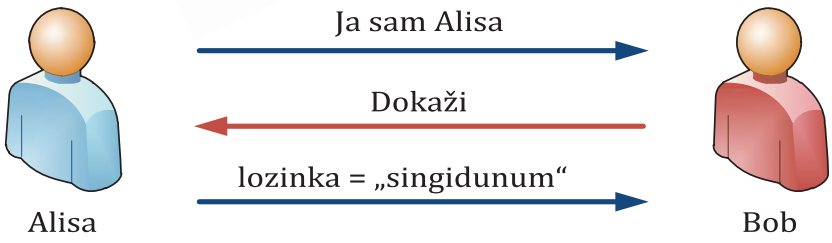
Upotrebom kriptografskih algoritama u Javi, lokalno ili preko mreže, simulirati sledeće kriptografske protokole. Za uspešnu realizaciju laboratorijskog rada, predlažemo upotrebu sledećih Java klasa (upotreba svih navedenih klasa je demonstrirana u udžbeniku):

- KeyGenerator
- SecretKey
- SecretKeySpec
- Cipher
- FileInputStream
- CipherInputStream
- FileOutputStream
- IvParameterSpec
- CertificateFactory
- ByteArrayInputStream
- Certificate
- DHParameterSpec
- AlgorithmParameterGenerator
- AlgorithmParameters
- KeyAgreement
- KeyPairGenerator
- KeyPair
- SecureRandom
- PrivateKey
- PublicKey
- Signature

- Path
- Files
- KeyStore
- RSAPrivateKey
- Mac
- MessageDigest
- SecretKeyFactory
- DESKeySpec
- PBEKeySpec
- Iterator
- X509CRL
- PKCS8EncodedKeySpec
- KeyFactory
- X509EncodedKeySpec
- Provider
- Key

**Zadatak 1:**

Protokol zahteva tri poruke između Alise i Boba. Alisa ima ulogu klijentskog računara, dok Bob ima ulogu serverskog računara. U prvoj poruci Alisa se predstavlja pred Bobom „Ja sam Alisa“, u drugoj poruci Bob traži od Alise da dokaže da je ona zaista Alisa. Alisa zna svoju lozinku „Singidunum“ i šalje je Bobu. Kod Boba se nalazi baza podataka koja sadrži lozinke svih svojih korisnika. Bob pretražuje Alisinu lozinku i nakon toga potvrđuje Alisin identitet. Simulirati protokol sa slike (Slika 0.1).




*Slika 0.1 Primer jednostavne autentifikacije*

**Zadatak 2:**

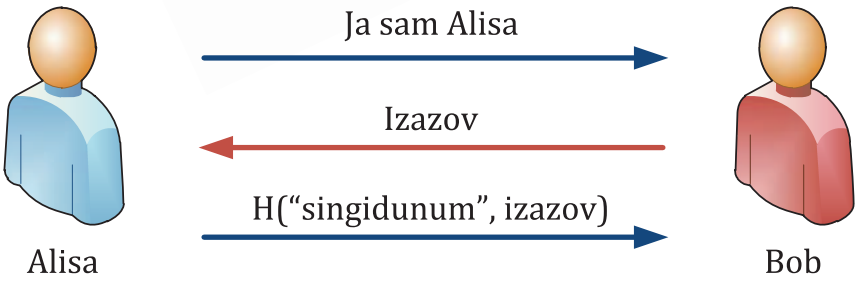
Komunikacija između Alise i Boba izgleda identično kao u prethodnom zadatku, osim u poslednjoj poruci kada Alisa šalje Bobu poruku tako što računa otisak poruke „*heš* od lozinke“ preko neke jednosmerne funkcije, kao što su *MD5*, *SHA1* i td. Na ovaj način Alisa krije lozinku od Boba i napadača. Da objasnimo, Bob sada u svojoj bazi podataka čuva samo Alisinu *heš* vrednost od lozinke, na osnovu koje na identičan način prihvata ili odbacuje Alisin identitet. Simulirati protokol sa slike (Slika 0.2).




*Slika 0.2 Poboljšani protokol za autentifikaciju*

**Zadatak 3:**

Ovde ćemo istaći da algoritmi koji se koriste za računanje *heš* vrednosti (*MD5*), ne spadaju u grupu kriptografskih tehnika, iz tog razloga kažemo da je uvođenjem kriptografskih tehnika moguće doći do još boljih rešenja koja će u velikoj meri doprineti podizanju nivoa bezbednosti kod protokola. Simulirati protokol sa slike (Slika 0.3), koji predstavlja modifikaciju prethodnog protokola.

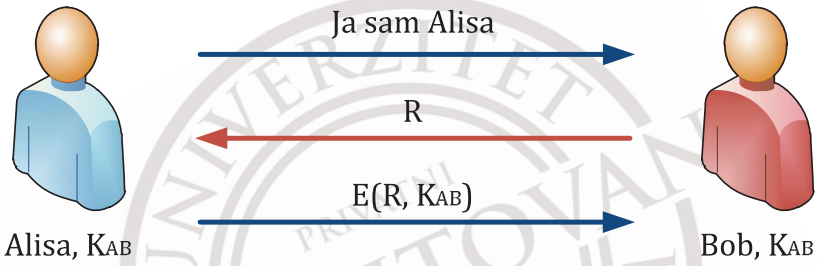



*Slika 0.3 Protokol za autentifikaciju, izazov – odgovor*

**Zadatak 4:**

Simulirati protokol za autentifikaciju simetričnim ključem. Sledi specifikacija protokola:

1. Alisa se predstavlja Bobu „Ja sam Alisa“.
2. Bob generiše slučajnu vrednost – izazov, zatim šalje Alisi.
3. Alisa šifruje sa simetričnim algoritmom slučajnu vrednost R, dobijeni šifrat šalje Bobu.
4. Bob dešifruje šifrat koji mu je Alisa poslala i verifikuje Alisu na osnovu izazova R koji je poslao u drugom koraku.

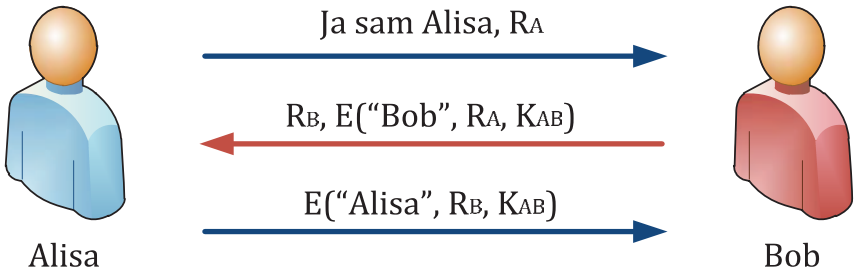



*Slika 0.4 Jednostrana autentifikacija simetričnim ključem*

**Zadatak 5:**

Simulirati protokol za uzajamnu autentifikaciju simetričnim ključem. Sledi specifikacija protokola:

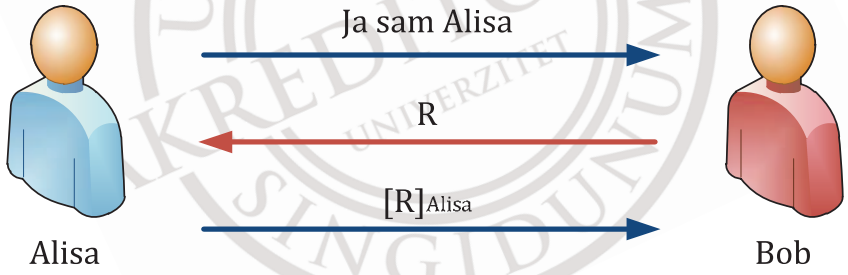
1. Alisa se predstavlja Bobu „Ja sam Alisa“ i generiše slučajnu vrednost $R_A$ – izazov, zatim šalje Bobu.
2. Bob šifruje sa simetričnim algoritmom i ključem $K_{AB,,}$ identitet parametar („Bob“) i slučajnu vrednost $R_A$ i generiše slučajnu vrednost $R_B$ za Alisu. Zatim dobijeni šifrat i vrednost $R_B$ šalje Alisi.
3. Alisa preko primljenog šifrata verifikuje Bobov identitet. Alisa šifruje vrednost $R_B$ i identitet parametar („Alisa“) ključem $K_{AB,}$ zatim generisani šifrat šalje Bobu na osnovu koga Bob verifikuje Alisin identitet.




*Slika 0.5 Uzajamna autentifikacija sa simetričnim ključem*

**Zadatak 6:**

Simulirati protokol sa slike (Slika 0.6). Protokol omogućava Bobu da autentifikuje Alisu preko javnog ključa, jer samo ona može da uradi operaciju šifrovanja sa svojim privatnim ključem i da prosledi korektan odgovor vrednost (dešifrovano R) u trećem koraku.



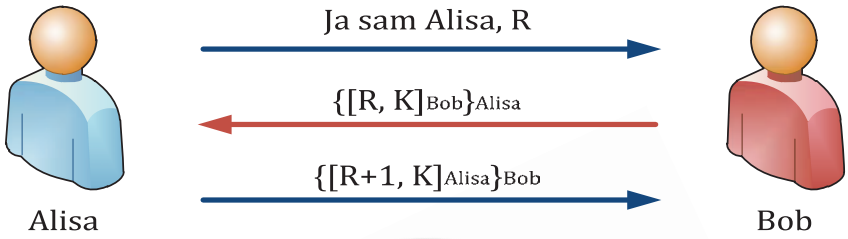

*Slika 0.6 Autentifikacija sa javnim ključem – šifrovanje*

Sledi specifikacija protokola:

1. Alisa se predstavlja Bobu „Ja sam Alisa".
2. Bob generiše slučajnu vrednost R (izazov) i šalje je Alisi.
3. Alisa u ovoj fazi koristi svoj privatni ključ za potpisivanje poruke (vrednost R - izazov) koju je dobila od Boba, zatim šalje potpisanu poruku R Bobu, na osnovu koga Bob verifikuje Alisu.

**Zadatak 7:**

Simulirati protokol sa slike (Slika 0.7). Protokol treba da omogućava uzajamnu autentifikaciju i razmenu sesijskog ključa upotrebom asimetrične kriptografije.

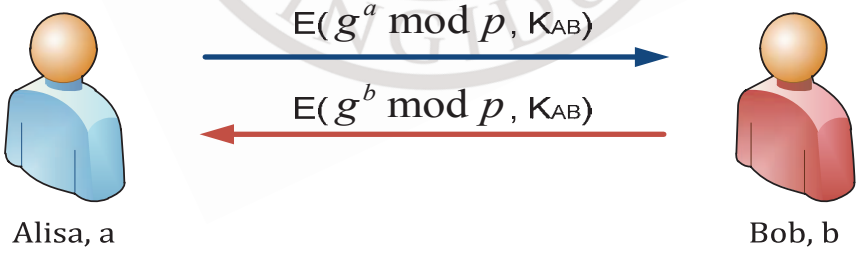



*Slika 0.7 Protokol za uzajamnu autentifikaciju i uspostavu sesijskog ključa – potpisivanje i šifrovanje javnim ključem*

Dizajn protokola treba da bude takav da obe strane prvo potpisuju vrednost R i ključ K sa privatnim ključem, a zatim šifruju sa javnim ključem druge strane.

**Zadatak 8:**

Simulirati protokol za razmenu sesijskog ključa preko PFS-a i simetričnih šifri. Dizajn protokola je prikazan na slici(Slika 0.8).



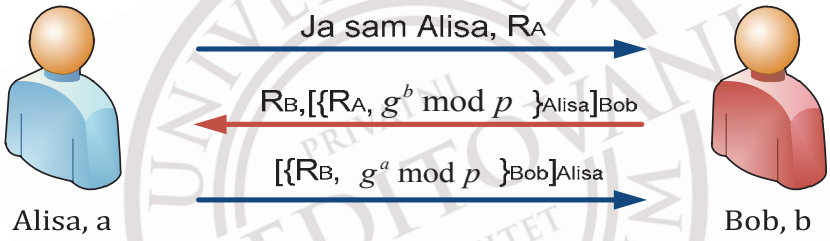

*Slika 0.8 Implementacija PFS metoda sa simetričnim šiframa*

Ovaj dizajn protokola je interesantan pristup za implementaciju *PFS* metode koja ispunjava sve postavljene zahteve. U ovoj realizaciji *PFS*-a, ključ sesije $K_S$ se više ne prenosi preko mreže. Možemo da primetimo da se $K_S$ sada izračunava na Alisinoj i

Bobovoj strani, a nakon izračunavanja Alisa i Bob permanentno zaboravljaju (uništavaju na siguran način) tajne parametre *a* i *b* koji su iskorišćeni samo za određeni ključ sesije. Za svaki novi ključ sesije, parametri će nanovo biti generisani na neki slučajan način. Ovim smo dobili efemerni *DH* protokol u kojem čak ni Alisa ni Bob ne mogu kasnije da rekonstruišu $K_S$.

**Zadatak 9:**

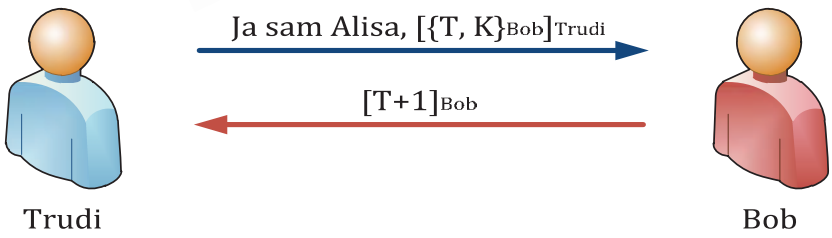
Simulirati protokol za razmenu sesijskog ključa preko PFS-a i asimetričnih šifri. Protokol sadrži sve elemente protokola sa kojim smo se do sada susretali. U ovom protokolu Alisa i Bob će biti autentifikovani. Komunikacija u vidu sesija biće bezbedna. Takođe, obezbeđen je *PFS*.




*Slika 0.9 Protokol za uzajamnu autentifikaciju, razmeni sesijskog ključa sa* PFS *metodom*

**Zadatak 10:**

Simulirati protokol sa vremenskim pečatom na slici (Slika 0.10). Traženi dizajn protokola sa vremenskim pečatom je u suštini verzija protokola potpiši pa šifruj.




*Slika 0.10 Autentifikacija sa vremenskim pečatom*

## LITERATURA

**A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray / S. Vo** A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications [Časopis]. - Gaithersburg : NIST, 2010.

Chapter 13: Encryption - Symmetric Key Agreement [Na mreži] // http://docstore.mik.ua/orelly/java-ent/security/ch13_07.htm. - O'Reilly & Associates, 2015.

**Flanagan David** Java™ in a Nutshell, Fourth Edition [Knjiga]. - [s.l.] : O'Reilly & Associates. - 0-596-00283-1.

**FlightRadar24** http://www.flightradar24.com/ [Na mreži]. - 2015.

**Gilstrap Brian R.** An Introduction to Cryptography and the Java Cryptography Extension.

Guidelines for Implementing and Using the NBS Data [Na mreži] // http ://www.itl.nist.gov/fipspubs/fip 74.htm. - FIPS 74.

**Huseby Sverre H.** Innocent Code - A Security Wake-Up Call for Web Programmers [Knjiga]. - [s.l.] : John Wiley & Sons Inc.

**M. Milosavljević, S. Adamović** Kriptologija 2 [Knjiga]. - Beograd : Univerzitet Singidunum, 2014. - 978-86-7912-537-8.

**M. Milosavljević, S. Adamović** Osnove teorije informacija i kodovanja [Knjiga]. - Beograd : Univerzitet Singidunum, 2014. - 978-867912-506-4.

**M. Tatović, S. Adamović, A. Jevremović, M. Milosavljević** One method for generating uniform random numbers via civil air traffic [Radovi] // SINTEZA - Singidunum University International Conference. - Beograd : [s.n.], 2014. - T. 1. - str. 606-609.

**M. Veinović, S. Adamović** Kriptologija 1 [Knjiga]. - Beograd : Univerzitet Singidunum, 2013.

**ORACLE** Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7 [Na mreži] // http://docs.oracle.com/. - 2014. - http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html.

Random [Na mreži] // http://www.random.org. - 2014.

**Schneier Niels Ferguson & Bruce** Practical Cryptography [Knjiga]. - ISBN 0471223573.

**Weiss Jason** Java cryptography extensions - practiical guide for programmers [Knjiga]. - San Francisco : [s.n.]. - T. 1.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

# REČNIK POJMOVA

## A

**AES -** *Advanced Encryption Standard*. Blokovska šifra sa simetričnim ključem, standardizovana od strane NIST-a. Pripada grupi modernih šifara. Dužina ključa je 128, 192, i 256 bitova.

**ASCII -** američki kodni standard za razmenu informacija. Ove kodove koristi veliki broj računarskih sistema za prevođenje karaktera u binarne brojeve.

**Algoritam -** opis za rešavanje nekog problema.

**Asimetrični šifarski sistemi –** sistemi sa javnim i privatnim ključem. Primer ovog tipa algoritama je *RSA* koji se koristi za šifrovanje i digitalno potpisivanje.

## B

**Biometrija -** tehnika za autentifikaciju koja koristi jedinstvene fizičke karakteristike nekog pojedinca. Postoje različiti tipovi biometrije.

**Biometrijski obrazac -** digitalizovana fizička karakteristika nekog pojedinca za određeni biometrijski izvor koji se koristi za autentifikaciju date osobe.

**Binarna sekvenca -** niz karaktera (jedinica i nula) ili binarni niz.

**Blok kod -** skupovi reči iste dužine koji predstavljaju strukturu red.

## V

**VOIP -** protokol koji se koristi za digitalizaciju govora, pakovanje i slanje preko *TCP*/*IP* veze.

**Vizuelna kriptografija -** specijalna tehnika šifrovanja koja omogućava skrivanje informacija na slici, na takav način da se može dešifrovati ljudskim okom bez upotrebe računara.

**Venona -** predstavlja One-time pad u realnosti. Koristila ga je sovjetska špijunska mreža koja je formirana na teritoriji *SAD* 1940-tih.

## G

**GSM -** globalni sistem za telekomunikacije. Internacionalni standard za satelitske telefone.

## D

**DES -** *Data Encryption Standard*. Blokovska šifra sa simetričnim ključem usvojena od strane *NIST*-a. Pripada grupi modernih šifara. Dužina ključa je 56 bitova, prevaziđen je od strane *AES*-a.

**Digitalni potpis -** protokol koji koristi primalac poruke da bi verifikovao pošiljaoca poruke. Za verifikaciju se koristi javni ključ pošiljaoca. Obezbeđuje servis neporecivosti.

**Dešifrovanje -** proces transformacije šifrata u otvoreni tekst.

**Digitalni vodeni pečat -** tehnika za zaštitu autorskih prava digitalnih sadržaja.

**Difi-Helman -** protokol namenjen za razmenu simetričnih ključeva preko javnih komunikacionih kanala.

**DSA -** asimetrični algoritam za digitalno potpisivanje.

## E

**Entropija -** mera za količinu neodređenosti ili prosečna količina informacije koju sadrže generisane poruke nekog informacionog izvora.

**ECB -** režim za šifrovanje kod blokovskih šifara u kom je svaki blok šifrata potpuno nezavisan. Ovaj režim može da bude nebezbedan.

## I

**Iris -** šara irisa ili obojeni deo oka koji je prilično haotičan (neodređen) ili slučajan. Predstavlja biometrijski izvor sa visokim performansama.

**Integritet -** prevencija od neautorizovane izmene sadržaja (poruke).

**Informacioni izvor -** generator sekvence simbola koji predstavljaju poruke.

## J

**Javni ključ -** element sistema sa javnim ključem ili javni ključ nekog korisnika koji je dostupan svim korisnicima u mreži. Koristi se u fazi šifrovanja.

**Jednosmerna funkcija -** sam naziv kaže da je funkcija neinvertibilna, ako je $y = f(x)$, tada je $x \neq f(y)$. Heš funkcije su vrsta jednosmernih funkcija.

## K

**Kapacitet kanala -** maksimalna prosečna količina informacija na izlazu kanala koja je jednaka količini informacije na ulazu.

**Kodna reč -** binarna sekvenca kojom je predstavljen simbol izvorne poruke.

**Kriptoanaliza -** umetnost i nauka o razbijanju šifrovanih poruka.

**Komunikacioni kanal -** formalizacija šta se dešava u prenetoj poruci između predajne i prijemne strane.

**Kompletan kod -** prefiksni kod čije kodno stablo ne sadrži nekorišćene listove.

**Kodovanje -** transformacija simbola poruke u kodne reči.

**Kriptološki ključ -** binarna sekvenca veće dužine, poseduje osobine slučajnih nizova. Koristi se kao početno stanje algoritma u fazi šifrovanja i dešifrovanja.

**Kriptograf -** lice koje dizajnira šifre.

**Kriptoanalitičar -** lice koje dizajnira metode za razbijanje šifara.

**Kriptografija -** nauka o pravljenju algoritama za šifrovanje.

**Kriptoanaliza -** nauka o razbijanju algoritama za šifrovanje.

## *L*

**LAN -** lokalna mreža.

**LFSR -** linearni pomerački registar sa povratnom spregom. Njegova uloga se sastoji u generisanju pseudo slučajnih sekvenci. Ima periodu (ponavlja se nakon izvesnog vremena).

**LSB -** steganografska tehnika za ugrađivanje tajnih poruka u podatak nosilac po principu zamene bitova sa najmanjim značajem.

**Lozinka -** tajni akreditiv (string od prosečno 8 karaktera) koji se koristi za autenitfikaciju kod tradicionalnih sistema za autentifikaciju. Korisnik je u stanju da zapamti vise lozinki, dok kriptološke ključeve nije moguće zapamtiti.

## *M*

**MAC -** kod za proveru autentičnosti poruke.

## N

**NIST -** američka federalna agencija za standarde i tehnologiju.

**NSA -** nacionalna agencija za bezbednost. Američka agencija koja je odgovorna za bezbednost i kriptoanalizu elektronskih komunikacija.

## O

**One-time pad -** perfektna šifra koja pripada grupi šifara sa simetričnim ključem. Ključ je generisan na slučajan način i ima jednaku dužinu kao i poruka. Ključ ne sme nikad da se ponavlja.

**Otvoreni tekst -** poruka u fazi šifrovanja ili rezultat funkcije za dešifrovanje.

## P

**PGP -** program za bezbednu elektronsku poštu koji objedinjuje servise za kriptografsku zaštitu podataka i digitalno potpisivanje. Razvijen od strane Fila Cimermana.

**PKI -** infrastruktura sa javnim ključevima. Sadrži bazu javnih ključeva preko kojih obezbeđuje autentifikaciju u srednjim i velikim mrežama.

**Privatni ključ -** element sistema sa javnim ključem ili tajna koju čuva svaki korisnik. Koristi se u fazi dešifrovanja.

**Protokol -** protokol je skup pravila koje razumeju i poštuju strane u komunikaciji.

**Poruka -** podatak koji se razmenjuje preko komunikasionog kanala između predajne i prijemne strane. Može da bude kodovana i u šifrovanoj formi.

**Perfektna tajnost** – teorijski dokazan kriptografski mehanizam koji ne zavisi od računarske snage.

**PRNG -** generator za generisanje pseudo slučajnih brojeva na osnovu inicijalnog stanja koje je određeno na slučajan način.

**PBE -** generisanje kriptološkog ključa na osnovu lozinke.

## R

**RSA -** kriptografski sistem sa javnim ključem, patentiran od strane Rivesta, Šamira i Adlemana 1976. Sigurnost *RSA* zasniva se na složenosti faktorizacije velikih brojeva. Ima široku primenu, implementiran je u *PGP* i *SSL*-u.

**RC4 -** sekvencijalna šifra sa simetričnim ključem. Pripada grupi modernih šifara. Dužina ključa jednaka je poruci. Neki je nazivaju pseudo perfektna šifra. Za generisanje ključa koristi se generator za pseudo slučajne brojeve. Pravila implementacije su skoro ista kao i kod perfektnih šifara.

## S

**Sertifikat -** elektronski fajl, obično sadrži javni ključ koji je digitalno potpisan od Sertifikacionog tela. Sertifikati se koriste za autentifikaciju na Internetu.

**Sertifikaciono telo -** treća strana od poverenja koja potpisuje i distribuira sertifikate.

**SSL -** široko primenjeni zaštitni protokol za autentivikaciju na Internetu (Internet transakcije)

**Simetrični ključ -** kriptološki ključ koji je identičan na predajnoj i prijemnoj strani.

**Steganografija -** nauka o skrivanju informacija u drugim bezazlenim podacima. Čest primer je skrivanje tekstualnog sadržaja u neku sliku.

**Stacionarni izvor** – informacioni izvor čija statistika ne zavisi od vremena.

**Sesijski ključ –** simetrični ključ za jednokratnu upotrebu.

## T

**TCP/IP -** protokol koji se koristi za prenos informacija u računarskim mrežama. TCP/IP je postao standard za sve mreže povezane sa Internetom.

**TDES -** blokovska šifra sa simetričnim ključem, usvojena od strane NIST-a. Bazirana na sukcesivnoj aplikaciji tri DES algoritma sa različitim ključevima. Predstavlja ojačanje klasičnog DES-a.

**TRNG -** generator slučajnih brojeva; ovakve generatore je moguće konstruisati u prirodnim informacionim izvorima.

## F

**Faktorizacija** - rastavljanje velikih prostih brojeva na činioce iz skupa prostih brojeva.

**Fejstel šifra -** nazvana po nemačkom kriptografu Horstu Fejstelu (Horst Feistel) koji je pionir u razvoju dizajna blokovskih šifara, radio je u IBM-u. Ovo su bila inicijalna istraživanja koja su kulminirala razvoju DES (Data Encryption Standard) algoritma 1970. godine. Fejstel šifra predstavlja dizajn blokovske šifre, a ne posebnu šifru.

## H

**Haker –** originalno se termin koristi za kreativnog programera koji razvija programe za ilegalan pristup računarskim mrežama u cilju krađe ili uništenja informacija. Dobar haker treba da poseduje dobre kriptoanalitičke veštine.

**Hamingovo rastojanje -** koristi se za određivanje razlike između dva niza jednake dužine.

**Heš funkcije -** funkcija koja generiše skraćenu verziju poruke. U kriptografskim aplikacijama heš funkcije moraju da budu jednosmerne sa malom verovatnoćom da dve različite poruke mogu da daju isti rezultat.

## C

**CRC -** Cyclic Redudancy Check. Kod široke primene, koristi se za detektovanje grešaka. Obično pre dešifrovanja, CRC se proverava.

**CBC -** režim za šifrovanje kod blokovskih šifara. Radi na principu ulančavanja blokova šifrata.

**CFB -** transformacija simetrične šifre iz blokovski u sekvencijalni režim šifrovanja.

## Š

**Šifra -** algoritam za šifrovanje. Mogu da postoje različiti tipovi šifara. Osnovne podele su na klasične i moderne, simetrične i asimetrične, apsolutno i praktično tajne.

**Šifrat -** rezultat funkcije za šifrovanje ili transformacija otvorene poruke koja je inverzna uz posedovanje pravog ključa.

**Šifrovanje -** proces transformacije otvorenog teksta u šifrat.

Na osnovu člana 23. stav 2. tačka 7. Zakona o porezu na dodatu vrednost („Službeni glasnik RS”, br. 84/2004, 86/2004 (ispr.), 61/2005, 61/2007 i 93/2012), Odlukom Senata Univerziteta Singidunum, Beograd, broj 260/07 od 8. juna 2007. godine, ova knjiga je odobrena kao osnovni udžbenik na Univerzitetu.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

007:004.056

АДАМОВИЋ, Саша, 1985-
Zaštita informacionih sistema / Saša Adamović. - 1. izd. - Beograd : Univerzitet Singidunum, 2015 (Loznica : Mobid). - 194 str. : ilustr. ; 24 cm

Na vrhu nasl. str.: Fakultet za informatiku i računarstvo. - Tiraž 400. -
Rečnik pojmova: str. 187-194. - Bibliografija: str. 185-186.

ISBN 978-86-7912-589-7

a) Информациони системи - Заштита
COBISS.SR-ID 212881932

© 2015.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.