ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

HARDVERSKA IMPLEMENTACIJA CAMELLIA ALGORITMA ZA ŠIFROVANJE SIMETRIČNIM KLJUČEM

– Master rad–

Kandidat:

Sanja Marković 2014/3366

Mentor:

doc. dr Zoran Čiča

Beograd, Septembar 2015.

SADRŽAJ

1. UVOD

Ubrzani razvoj savremenih tehnologija poslednjih godina otvorio je vrata stvaranju novih uređaja, sistema i usluga koje se stavljaju na raspolaganje korisniku i imaju za cilj olakšavanje i ubrzavanje obavljanja svakodnevnih aktivnosti. Jedno od najbrže rastućih tržišta je tržište mobilnih uređaja i propratne opreme, jer su oni postali neizostavni deo inventara većine ljudi. Po istraživanjima (2014) čak 74% ljudi koristi barem neku od trenutno prisutnih socijalnih mreža. Od toga blizu 50% njih to čini preko mobilnih telefona, a 20% preko tableta. Osim socijalnih mreža i elektronsko bankarstvo je iskoristilo sve pogodnosti mobilnih uređaja i njihove popularnosti, tako da trenutno oko 40% korisnika koji poseduju i mobilni telefon i račun u banci, koristi mobilni telefon za vršenje transakcija, plaćanje računa itd. Zahvaljujući razvoju RFID i NFC tehnologije, platne i kreditne kartice sve više bivaju zamenjene mobilnim uređajima u kojima su navedene tehnologije zastupljene.

Međutim, faktor koji se nikako ne sme zanemariti je bezbednost korisnika, čuvanje njegovog identiteta i tajnosti podataka, onemogućavanje trećih lica da vide i modifikuju sadržaj poruka koje se šalju preko mreže itd. Ovo je jedan od najvećih izazova kada su u pitanje digitalne komunikacije. Glavni problem je u tome što su svi korisnici širom sveta povezani na istu mrežu tj. Internet. Stoga se kao prioritet postavlja bezbednost korisnika tj. njegovih podataka i podataka koje on šalje. Kako bi se podaci koji putuju kroz mrežu zaštitili od trenutka slanja do trenutka prijema koriste se razne kriptografske metode, o kojima će kasnije biti više reči.

Ovaj rad se bavi hardverskom implementacijom Camellia algoritma za enkripciju. Realizovani algoritam se može koristiti u različitim uređajima za kriptovanje poruka, sadržaja paketa i dr. Za realizaciju implementacije koristi se VHDL programski jezik, a razvoj i verifikacija dizajna vrši se u ISE razvojnom okruženju za FPGA čipove proizvođača Xilinx.

Ostatak rada organizovan je na sledeći način. Drugo poglavlje daje osnovne pojmove o kriptografiji, dat je pregled najpoznatijih algoritama i njihova primena. Treće poglavlje opisuje strukturu Camellia algoritma, definisane su celine koje čine ovaj algoritam. Četvrto poglavlje bavi se opisivanjem realizovane implementacije, najpre su definisani interfejsi entiteta, a zatim su detaljno objašnjene komponente od kojih je formiran algoritam. U prvom delu petog poglavlja je prikazana verifikacija dizajna, u drugom delu prikazana je analiza performansi realizovanog rešenja, upotrebljeni resursi na čipu i maksimalna frekvencija na kojoj dizajn može da radi. Poslednje, šesto poglavlje, sadrži zaključna razmatranja autora ove teze.

2. OSNOVNI POJMOVI KRIPTOGRAFIJE

Kao što je već pomenuto u uvodu ove teze, Internet može biti veoma nebezbedno mesto. U samom početku postojanja računarskih mreža tj. Interneta, on je pretežno bio korišćen od strane naučnika i istraživača u okviru raznih instituta i Univerziteta. Međutim, danas je on dostupan svima, pa se među korisnicima često mogu naći oni čiji je primarni cilj ostvarivanje nekog vida lične koristi na nedozvoljen način. Stoga su bezbednosni zahtevi u današnje vreme daleko komplikovaniji.

Bezbednosni problemi u mreži se mogu razvrstati u četiri grupe[2]:

- Tajnost (poverljivost): podrazumeva da neovlašćena lica ne mogu mogu doći do podataka
- Provera identiteta učesnika (autentifikacija): predstavlja postupak u kome se proverava sa kime se stupa u kontakt i da li je ta osoba zaista ona za koju vam se predstavlja;
- Isključivanje poricanja: je najbitnije u sferi E-komerca tj. E-kupovine gde je veoma bitno da potencijalni kupac ili prodavac ne može u nekom trenutku tvrditi da je npr. poručio manju količinu robe ili po nekoj drugoj ceni od stvarne.
- Kontrola integriteta podataka: Ona mora da osigura da poruka koja je poslata ni u jednom trenutku nije kompromitovana i izmenjivana od strane trećeg lica.

Sistemi za zaštitu se mogu implementirati na različitim mrežnim slojevima.

- Tako se na fizičkom sloju oprema štiti fizički. Najčešće nekim vidom mehaničke zaštite od pristupa, npr. vodovi se štite hermetički zatvorenim cevima u kojima se nalazi komprimovani vazduh pa se pri svakoj nagloj promeni pritiska aktivira alarm kao upozorenje.
- U sloju linka za podatake primenjuje se "šifrovanje linije". Ono podrazumeva da se podaci na jednom kraju šifruju, a potom na drugom dešifruju. Međutim, ukoliko podaci treba da prođu kroz veliki broj mrežnih uređaja (rutera, svičeva), oni se na svakom od njih moraju dešifrovati, pa podaci postaju podložni napadima iz samog usmerivača. Zbog toga što se u mrežnom uređaju poruke dešifruju u originalni oblik, u slučaju da napadač ima pristup mrežnom uređaju, bezbednost može biti narušena. Takođe, jedan od nedostataka je taj što se ne može birati koje se poruke šifruju, već će svaka sesija biti šifrovana. Ovakva vrsta zaštite koristi se u slučajevima kada je sam medijum za prenos nebezbedan (npr. vazduh kod bežičnih komunikacija, fizički neobezbeđen link između dva mrežna uređaja).
- U mrežnom sloju se postavljaju zaštitne barijere koje u mrežu propuštaju "dobre", a zadržavaju "loše" pakete.
- U transportnom sloju postoji mogućnost šifrovanja kompletnih sesija s kraja na kraj. Ako se zahteva maksimalna bezbednost, ova implementacija je neizostavna.
- Na sloju aplikacija se mogu vršiti autentifikacija, šifrovanje i isključivanje poricanja.

- Izuzev zaštite na fizičkom sloju svi ostali vidovi zaštite se zasnivaju na kriptografiji.

2.1. Kriptografija

Predstavlja veštinu smišljanja i formiranja šifrovanih poruka tj. nauku koja se bavi metodama i principima zaštite tajnosti podataka. Veština "razbijanja" šifara naziva se kiptoanaliza. Kriptografija i kriptoanaliza zajedno formiraju naučnu disciplinu koja se naziva kriptologija.

U okviru kriptografije mogu se razlikovati dva različita principa skrivanja značenja poruka:

- Šifrovanje
- Kodiranje.

2.2. Šifrovanje

Šifrovanje predstavlja zamenu "znak za znak" (bit za bit) ne obraćajući pažnju na jezičko značenje poruke. Poruka čiju tajnost želimo da sačuvamo, a koja se se šalje preko "nesigurnog kanala" naziva se "otvoreni tekst" (*plain text*). Da bi poruku šifrovali vršimo njenu transformaciju korišćenjem funkcije čiji je parametar "ključ". Kao rezultat dobijamo izmenjenu poruku tj. "šifrovani tekst".

U ranijoj istoriji koristile su se razne metode šifrovanja kao što su, supsticione šifre koje su menjale znak ili grupu znakova drugim znakom, potom transpozicione šifre podrazumevaju menjanje redosleda znakova itd.

2.3. Algoritmi za šifrovanje

Za razliku od prošlih vremena kada se kriptovanje poruka zasnivalo na jednostavnim algoritmima, savremena kriptografija za cilj ima stvaranje složenih algoritama koji će onemogućiti neželjeno dešifrovanje poruka bez obzira na raspoložive resurse. Algoritmi za šifrovanje mogu se realizovati na dva načina:

- Hardverska realizacija: omogućava daleko veću brzinu obrade podataka za šifrovanje
- Softverska realizacija: obezbeđuje faktički neograničenu fleksibilnost

Zavisno od načina korišćenja ključa, razvile su se dve klase algoritama:

- Algoritmi za šifrovanje simetričnim ključem i
- Algoritmi za šifrovanje asimetričnim ključem

2.3.1. Algoritmi za šifrovanje simetričnim ključem

Ove algoritme delimo u dve grupe: stream šifrovanje i blok šifrovanje.

Stream šifrovanje funkcioniše tako što se enkripcija vrši bit po bit, dok se kod blok šifrovanja enkripcija vrši po blokovima podataka tj. uzimaju se blokovi od više bita (64, 128, 196,

256...) i kao takva celina se šifruju. Dešifrovanje se najčešće vrši obrnutim šifrovanjem, algoritam je isti samo se ključevi za šifrovanje koriste obrnutim redosledom.

U daljem tekstu navešćemo neke od algoritama koji se zasnivaju na principu šifrovanja sa simetričnim ključem među kojima se nalazi i Camellia koja je predmet ovog rada.

i) DES (Data Encription Standard)

DES predstavlja jedan od prvih široko prihvaćenih kombinovanih algoritama za šifrovanje, zahvaljujući tome što ga je Američka Vlada usvojila kao zvanični standard za zaštitu informacija od javnog značaja. Algoritam je razvio IBM 1977. godine. Osnova ovog algoritma je Lucifer algoritam koji je takođe projektovan od strane IBM-a. Usled mnogih kontroverzi oko postojanja namerno ostavljenog backdoor-a i postojanja algoritama koji omogućavaju razbijanje šifre, DES više nije u upotrebi.

Princip rada se zasniva na tome što se otvoreni tekst šifruje u blokovima od 64 bita. Kao rezultat dobija se 64-bitni blok šifrovanog teksta. Ulazni parametar algoritma za šifrovanje koji ima 19 nezavisnih koraka je 56-bitni ključ. Prvi korak je transponovanje otvorenog teksta koje se obavlja bez upotrebe ključa, dok je poslednji korak inverzan prvom. U pretposlednjem koraku prva 32 bita menjaju mesto za zadnja 32 bita. Ostalih 16 koraka su sa funkcionalne strane identični, ali su vrednosti njihovih parametara rezultati primene raznih funkcija na prvobitni ključ. Pri dešifrovanju treba primeniti iste korake samo obrnutim redosledom.

ii) Triple DES

Usled prevaziđenosti DES algoritma tj. njegove jednostavnosti, IBM je došao do načina da ga učini složenijim. Tako je nastao Trostruki DES. Kao što se iz samog imena može zakjučiti on se zasniva na trostrukom šifrovanju čime je efektivno povećana dužina DES algoritma.

Ova metoda se odvija u tri koraka korišćenjem dva ključa. U prvom koraku se primenjuje šifrovanje standardnim DES algoritmom korišćenjem ključa K1. U sledećem koraku se koristi DES ali u režimu dešifrovanja uz pomoć ključa K2. Treći korak je ponovno šifrovanje pomoću ključa K1.

Razlog za korišćenje ŠIFROVANJA-DEŠIFROVANJA-ŠIFROVANJA (EDE), umesto ŠIFROVANJA-ŠIFROVANJA-ŠIFROVANJA (EEE) je kompatibilnost unazad sa običnim DES algoritmom. To znači da čak i ako jedan korisnik koristi Triple DES, a drugi DES između njih se može ostvariti komunikacija tako što će korisnik koji koristi TDES postaviti vrednost ključeva na K1=K2.

iii) AES (Advanced Encription Standard)

Kada je DES, a potom i Triple DES postao nedovoljno siguran, NIST (*National Institute of Standards and Technology*) se u saradnji sa Vladom SAD odlučio za raspisivanje javnog konkursa (nadmetanja) za novi standard za enkripciju koji bi se zvao AES. Pravila i zahtevi konkursa su bili precizno definisani. Glavni faktori za odabir bili su bezbednost, efikasnost, jednostavnost, fleksibilnost i memorijski zahtevi. Za pobednika konkursa odabrani su Joan Daemen i Vincent Rijmen i njihov algoritam imena "RIJNDAEL", koji je kovanica prezimena autora.

iv) Rijndael

Ovaj algoritam podržava blokove i ključeve veličina od 128 do 256 bita, u koracima od po 32 bita. Dužina bloka i dužina ključa su međusobno nezavisni. Ali pošto je jedan od zahteva konkursa za AES bio da algoritam mora da podržava ključeve od 128, 192 i 256 bita, i da je veličina bloka 128 bita, tako se AES realizuje u dve izvedbe:

- 128-bitni blok i 128-bitni ključ
- 128-bitni blok i 256-bitni ključ

v) Ostali algoritmi

Neki od postojećih ređe korišćenih algoritama sa šifrovanjem simetričnim ključem.

- RC4 (1-2048 bita)
- RC5 (128-256 bita)
- Serpent (128-256 bita)
- Twofish (128-256 bita)

Ovi algoritmi se mogu međusobno kombinovati (i sa DES I AES), po principu prvo se šifruje jednim algoritmom, a potom drugim.

2.3.2. Algoritmi za šifrovanje asimetričnim ključem

Distribuiranje ključeva oduvek je bilo najslabiji element kriptosistema. Ukoliko dođe do krađe ključa, sistem više nije neprobojan. Stoga su se dva istraživača sa Stanforda dosetila da naprave novu vrstu sistema sa različitim ključevima za šifrovanje i dešifrovanje koji se ne mogu lako izvesti jedan iz dugog. Ovi algoritmi se još nazivaju i algoritmi sa javnim ključem.

Njihov predlog je bio da algoritam za šifrovanje E i algoritam za dešifrovanje D treba da ispune sledeća tri zahteva:

1. D(E(P))=P.
2. D se izuzetno teško može izvesti iz E.
3. E se ne može provaliti šifrovanjem izabranog otvorenog teksta.

Prvi zahtev znači da primenom algoritma D na šifrovanu poruku E(P) treba da dobijemo originalni otvoreni tekst poruke P. Ukoliko ovaj zahtev nije ispunjen, primalac neće moći da dešifruje poruku. Drugi zahtev je jasan, dok je treći neophodan, jer uljez može da bude uporan u eksperimentisanju sa algoritmom ali bezuspešno.

Za šifrovanje javnim ključem korisnik mora da ima javni i privatni ključ. Javni ključ koristi svako ko želi da tom korisniku šalje šifrovane poruke, a privatni ključ služi korisniku za dešifrovanje primljenih poruka.

Poznati algoritmi koji ispunjavanju navedena tri zahteva su:

- RSA

Za šifrovanje su potrebni parametri e i n a za dešifrovanje d i n. Oni se određuju na sledeći način:

1. Izaberu se dva velika prosta broja, p i q (najčešće od po 1024 bita)
2. Izračuna se $n=p\times q$ i $z=(p-1)\times(q-1)$
3. Izabere se broj koji je prost u odnosu na z i označi se sa d
4. Pronađe se takvo e da zadovolji relaciju $e\times d = 1 \bmod z$

Javni ključ je sada (n,e), a tajni (n,d).

Šifra c se dobija na sledeći način:

$$c = m^e \bmod n$$

Izvorna poruka se dobija iz relacije:

$$m = c^d \bmod n$$

Algoritam RSA je suviše spor za šifrovanje veće količine podataka, ali se zato naširoko koristi za distribuiranje ključeva koji se zatim koriste u algoritmima za šifrovanje simetričnim ključem, kao što su AES i trostruki DES.

- Algoritam „ranca“

Osnovna ideja ovog algoritma je da neko ima veliki broj predmeta, svaki različite težine. Vlasnik predmeta kodira poruku tako što tajno bira određen podskup predmeta i stavlja ih u ranac. Ukupna težina predmeta u rancu se objavljuje, kao i spisak svih postojećih predmeta. Sadržaj ranca se drži u tajnosti. Otkrivanje liste predmeta u rancu poznate težine se smatra, uz još neka ograničenja, kao nerešiv problem, pa je to dalo ideju za ovaj algoritam.

- Razni algoritmi koji se baziraju na eliptičkim funkcijama

3. CAMELLIA ALGORITAM

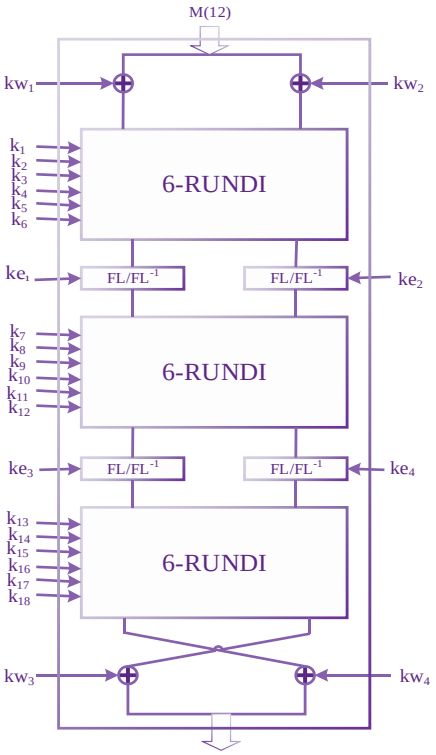
Camellia algoritam je algoritam za šifrovanje simetričnim ključem i spada u grupu blok šifara. Ovaj algoritam procesira blokove podataka od 128 bita sa tajnim ključem dužina 128, 192 ili 256 bita. Treba napomenuti da Camellia algoritam ima isti interfejs kao AES. U našoj implementaciji bavićemo se algoritmom koji koristi ključ od 128 bita.

Korišćenjem ključa ove dužine dobijamo Feistel simetričnu strukturu za formiranje blok šifre od 18 rundi. Nakon šeste i dvanaeste runde primenjuje se FL odnosno FL^{-1} funkcije kako bi omogućile neregularnost između rundi. Ove funkcije bi trebalo da obezbede dodatnu sigurnost u slučaju budućih napada.

Camellia algoritam može se podeliti na dve celine: *data randomizing* i *key scheduling* [2].

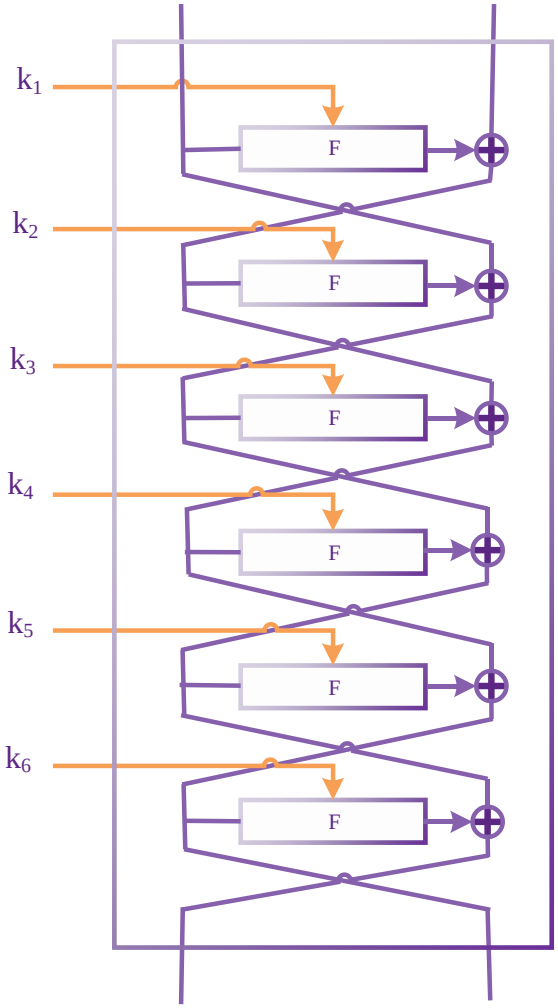
3.1. *Data randomizing* segment

Data_randomozing segment čini Feistel struktura od 18 rundi, pri čemu je posle šeste i dvanaeste runde ubačen FL/FL^{-1} sloj. Na samom početku kao i na kraju vrši se *pre-whitening* odnosno *post-whitening* tehnika koja se uvodi kako bi se povećala sigurnost blok šifara. *Key scheduling* segment, koji će u kasnijem delu teze biti objašnjen generiše ključeve za svaku od rundi, FL i FL^{-1} , kao i za *pre-* i *post-whitening*. Struktura *data randomizing* entiteta data je na slici 3.1.1.



Slika 3.1.1. Struktura entiteta *data_randomizing*

U okviru data_randomizing komponente izdvaja se celina koja se sastoji od 6 identičnih rundi. U svakoj rundi se poziva F-funkcija i unosi novi ključ. Ova celina se se ponavlja tri puta, stoga je od tih 6 rundi formirana zasebna komponenta koju smo nazvali *sest_rundi*. Unutrašnja struktura komponente *sest_rundi* data je na slici 3.1.2.



Slika 3.1.2. Struktura komponente *sest_rundi*

FL funkcija je definisana na sledeći način [3]:

$$Y_{R(32)} = \left(\left(X_{L(32)} \cap kl_{L(32)}\right) <<< 1\right) \oplus X_{R(32)} \quad (3.1.1)$$

$$Y_{L(32)} = \left(Y_{R(32)} \cup kl_{R(32)}\right) \oplus X_{L(32)} \quad (3.1.2)$$

gde simbol $\cap$ predstavlja operaciju „I“, simbol $\cup$ operaciju „ILI“, $\oplus$ predstavlja operaciju ekskluzivno „ILI“, a simbol $<<<$ ciklično rotiranje u levo za n bita.

Pri čemu Y_(L(32)) predstavlja prvih 32 bita (*most significant bits*) 64-bitnog izlaza a Y_(R(32)) drugih 32 bita (*least significant bits*).

FL^{-1} funkcija je samo inverzna funkcija funkcije FL.

F-funkcija je definisana na sledeći način [3]:

$$Y_{64} = P(S(X_{64} \oplus k_{64}) \quad (3.1.3)$$

P-funkcija je formirana samo sa XOR komponentama i linearna je transformacija ulaza od 8 bajtova u izlaz od takođe 8 bajtova.

S-funkcija je funkcija koja vrši zamenu koristeći jednu od 4 s-kutije. Sve s-kutije su povezane i definisane sledećim relacijama [3]:

$$S_1(x) = h(g(f(x \oplus a))) \oplus b \quad (3.1.4)$$

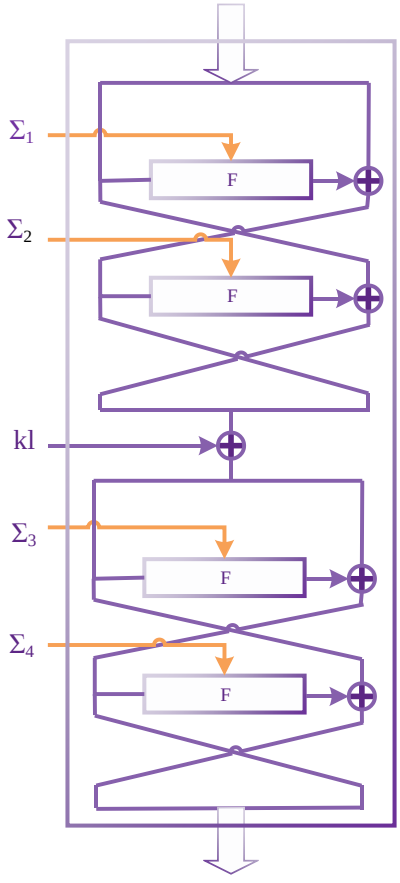
$$S_2(x) = S_1(x) <<< 1 \quad (3.1.5)$$

$$S_3(x) = S_1(x) >>> 1 \quad (3.1.6)$$

$$S_4(x) = S_1(x <<< 1) \quad (3.1.7)$$

3.2. *Key Scheduling* segment

Key scheduling segment za 128-bitni ključ formira 26 64-bitnih ključeva koji se koriste u 18 rundi, *pre-* i *post-whitening*-u i FL odnosno FL^{-1} funkcijama. Od 128-bitnog ključa *k* formiraju se *kl* i *kr*, pri čemu je *kl*=*k*, a *kr*=0, a potom se pomoću njih formira promenljiva *ka*. Formiranje *ka* prikazano je na slici 3.2.1. $\sum_{i(64)}(i = 1,2,3,4)$su 64-bitne konstante [4].



Slika 3.2.1. Struktura komponente *generisanje_ka*

Vrednosti konstanti korišćenih pri formiranju *ka* date su u tabeli 3.2.1.

Tabela 3.2.1. Konstante koje se koriste za formiranje promenljive *ka*

$\Sigma_{1(64)}$	0xA09E667F3BCC908B
$\Sigma_{2(64)}$	0xB67AE8584CAA73B2
$\Sigma_{3(64)}$	0xC6EF372FE94F82BE
$\Sigma_{4(64)}$	0x54FF53A5F1D36F1C
$\Sigma_{5(64)}$	0x10E527FADE682D1D
$\Sigma_{6(64)}$	0xB05688C2B3E6C1FD

Cikličnim rotiranjem u levo promenljivih *ka* ili *kl* za 15 ili 17 bita, formiraju se ključevi za svaku od rundi.

U tabeli 3.2.2. prikazano je formiranje svih 26 ključeva.

Tabela 3.2.2. Način formiranja ključeva u okviru komponente *key_scheduling*

	Ključ	Vrednost
Prewhitening	$kw_{1(64)}$ $kw_{2(64)}$	$(kl<<<_0)_{L(64)}$ $(kl<<<_0)_{R(64)}$
F(Runda 1) F(Runda 2) F(Runda 3) F(Runda 4) F(Runda 5) F(Runda 6)	$k_{1(64)}$ $k_{2(64)}$ $k_{3(64)}$ $k_{4(64)}$ $k_{5(64)}$ $k_{6(64)}$	$(ka<<<_0)_{L(64)}$ $(ka<<<_0)_{R(64)}$ $(kl<<<_{15})_{L(64)}$ $(kl<<<_{15})_{R(64)}$ $(ka<<<_{15})_{L(64)}$ $(ka<<<_{15})_{R(64)}$
FL FL^{-1}	$kl_{1(64)}$ $kl_{2(64)}$	$(ka<<<_{30})_{L(64)}$ $(ka<<<_{30})_{R(64)}$
F(Runda 7) F(Runda 8)	$k_{7(64)}$ $k_{8(64)}$	$(kl<<<_{45})_{L(64)}$ $(kl<<<_{45})_{R(64)}$

F(Runda 9) F(Runda 10) F(Runda 11) F(Runda 12)	$k_{9(64)}$ $k_{10(64)}$ $k_{11(64)}$ $k_{12(64)}$	$(ka<<<_{45})_{L(64)}$ $(ka<<<_{45})_{R(64)}$ $(ka<<<_{60})_{L(64)}$ $(ka<<<_{60})_{R(64)}$
FL FL^{-1}	$kl_{3(64)}$ $kl_{4(64)}$	$(ka<<<_{77})_{L(64)}$ $(K_L<<<_{77})_{R(64)}$
F(Runda 13) F(Runda 14) F(Runda 15) F(Runda 16) F(Runda 17) F(Runda 18)	$k_{13(64)}$ $k_{14(64)}$ $k_{15(64)}$ $k_{16(64)}$ $k_{17(64)}$ $k_{18(64)}$	$(kl<<<_{94})_{L(64)}$ $(kl<<<_{94})_{R(64)}$ $(ka<<<_{94})_{L(64)}$ $(ka<<<_{94})_{R(64)}$ $(kl<<<_{111})_{L(64)}$ $(kl<<<_{111})_{R(64)}$
Postwhitening	$kw_{3(64)}$ $kw_{4(64)}$	$(ka<<<_{111})_{L(64)}$ $(ka<<<_{111})_{R(64)}$

4. IMPLEMENTACIJA CAMELLIA ALGORITMA

U ovom poglavlju biće opisan programski kod korišćen za hardversku implementaciju Camellia algoritma. Za realizaciju implementacije korišćen je VHDL programski jezik.

U nastavku ovog poglavlja najpre će biti opisani interfejsi dizajna, a zatim komponente dizajna kao i njegova celokupna struktura.

4.1. Interfejsi

Dizajn sadrži ulazne i izlazne interfejse. Ulazni interfejsi su *data_in*, *kljuc* i *clk*, a izlazni je *data_out*. Sledi opis ovih interfejsa:

Signal *data_in* predstavlja ulazni podatak, tj. otvoreni tekst širine 128 bita koji treba šifrovati.

Signal *clk* se koristi kao signal takta. Uzlazna ivica ovog takta se koristi za početak izvršavanja naredbi u kodu.

Signal *kljuc* predstavlja 128-bitni ključ kojim se šifruje osnovni tekst.

Signal *data_out* je izlazni podatak širine 128 bita i predstavlja konačni rezultat šifrovanja.

4.2. Komponente dizajna

Pri implementaciji dizajna moramo najpre objasniti njegove najbitnije komponente :

- F-funkcija
- FL- funkcija
- FL^{-1} – funkcija
- S- funkcija
- P-funkcija

F-funkcija koju smo definisali u prethodnom poglavlju uzima dva parametra. Jedan je 64-bitni ulaz *f_in*, a drugi je 64-bitni ključ *ke*. Kao rezultat funkcija vraća 64-bitni rezultat *f_out*.

Najpre se vrši XOR operacija nad ulaznim parametrima. Potom se izvršava S-funkcija koja predstavlja supstituciju koristeći četiri s-kutije. S-kutije predstavljaju zasebne komponente čije instance pozivamo u okviru F-funkcije. Kod koji to izvršava je sledeći:

```
x<= f_in xor ke;
t1<=x(0to7);
t2<=x(8to15);
t3<=x(16to23);
t4<=x(24to31);
t5<=x(32to39;
t6<=x(40to47);
t7<=x(48to55);
```

```
t8<=x(56to63);

S1a : sbox1
portmap(
        sbox1_in => t1,
        sbox1_out => t01
        );

S1b : sbox1
portmap(
        sbox1_in => t8,
        sbox1_out => t08
        );

S2a : sbox2
portmap(
        sbox2_in => t2,
        sbox2_out =>t02
        );

S2b : sbox2
portmap(
        sbox2_in => t5,
        sbox2_out => t05
        );

S3a : sbox3
portmap(
        sbox3_in => t3,
        sbox3_out =>t03
        );

S3b : sbox3
portmap(
        sbox3_in => t6,
        sbox3_in => t06
        );

S4a : sbox4
portmap(
        sbox4_in => t4,
        sbox4_out_in => t04
        );

S4b : sbox4
portmap(
        sbox4_in => t7,
        sbox4_out => t07
         );
```

Konačno, poziva se P-funkcija koja ulaz od 8 bajtova linearno transformiše u izlaz od 8 bajtova. Kod koji to izvršava je sledeći:

```
y(0to7)<=t01 xor t03 xor t04 xor t06 xor t07 xor t08;
y(8to15)<=t01 xor t02 xor t04 xor t05 xor t07 xor t08;
y(16to23)<=t01 xor t02 xor t03 xor t05 xor t06 xor t08;
y(24to31)<=t02 xor t03 xor t04 xor t05 xor t06 xor t07;
y(32to39)<=t01 xor t02 xor t06 xor t07 xor t08;
```

```
y(40to47)<=t02 xor t03 xor t05 xor t07 xor t08;
y(48to55)<=t03 xor t04 xor t05 xor t06 xor t08;
y(56to63)<=t01 xor t04 xor t05 xor t06 xor t07;
```

FL i FL^{-1}-funkcije uzimaju dva parametra. FL funkcija za ulaz uzima 64-bitni parametar *fl_in* i 64-bitni ključ *fl_k*, izlazni podatak će biti 64-bitni *fl_out*. Ulazni podatak *fl_in* deli se na dve celine od 32 bita, pri čemu jednu celinu čine prva 32 bita ulaznog podatka i označeni su kao *x1*, dok je drugu celinu čine druga 32 bita i oni su označeni kao *x2*. Na isti način se deli i ključ *fl_k* na parametre *k1* i *k2*.

Izlazni podatak *fl_out* dobija se konkatanacijom promenljivih *d1* i *b1*.

Promenljiva *d1* se dobija tako što se najpre izvrši AND operacija nad parametrima *x1* i *k1*, a zatim se tako dobijeni rezultat ciklično pomeri u levo za jedan bit i konačno se nad tim rezultatom i parametrom *x1* izvrši XOR operacija.

Promenljiva *b1* dobija se tako što se izvrši OR operacija nad parametrima *x2* i *k2*, a zatim se nad tim rezultatom i parametrom *x1* izvrši operacija XOR.

Kod koji to izvršava je sledeći:

```
--FL funkcija
   x1<= fl_in(0to31);
   x2<= fl_in(32to63);
   k1<= fl_k(0to31);
   k2<= fl_k(32to63);
   a1<= x1 and k1;
   b1<= x2 xor(a1(1to31)&a1(0));
   c1<= b1 or k2;
   d1<= x1 xor c1;
   fl_out<= d1 & b1;
```

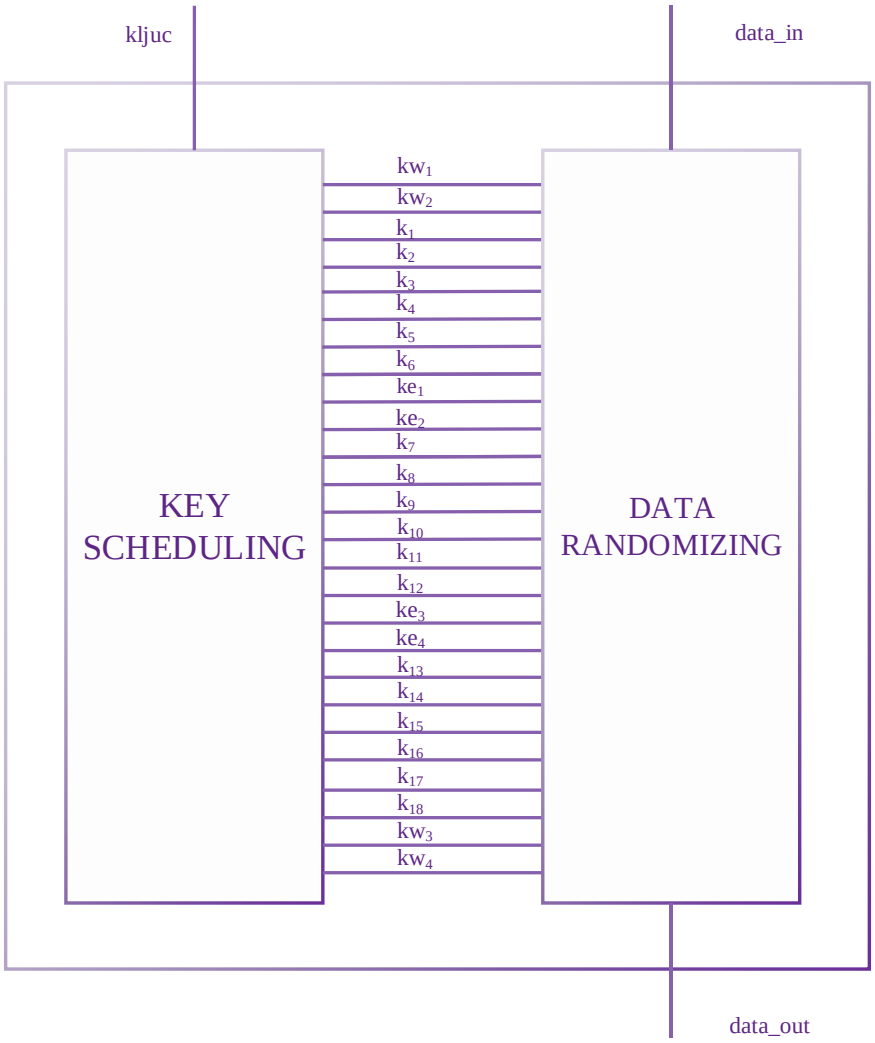
Funkcija FL^{-1} predstavlja inverznu funkciju funkcije FL, ulazni parametri koji se koriste su 64-bitni *fli_in* i 64-bitni ključ *fli_k*, dok je izlazni 64-bitni parametar označen kao *fli_out*.

Kod koji prikazuje ovu funkciju je sledeći:

```
--FLINV funkcija
   y1<= fli_in(0to31);
   y2<= fli_in(32to63);
   k1i<= fli_k(0to31);
   k2i<= fli_k(32to63);
   a2<= y2 or k2i;
   b2<= y1 xor a2;
   c2<= b2 and k1i;
   d2<= y2 xor(c2(1to31)&c2(0));
   fli_out<= b2 & d2;
```

4.3. Struktura implementacije

Dizajn se sastoji iz dva dela, *data_randomizing* komponente koja vrši enkripciju koristeći Feistel strukturu od 18 rundi zajedno sa FL i FL^{-1} funkcijama ubačenih na svakih 6 rundi i *key_scheduling* komponente koja formira ključeve neophodne za svaku od rundi. Ova dva dela realizovana su kao kombinaciona logika. U sinhronom procesu top level entiteta vrši se prosleđivanje ulaznog podatka i ključa, sinhrono na ivicu takta, na ulaze ovih komponenti, a rezultat šifrovanja se preuzima sa izlaza data_randomizing komponente. Na slici 4.3.1. su prikazane ove dve komponente i način na koji su povezane.



Slika 4.3.1. Struktura Camellia entiteta koji čine data_randomizing i key_scheduling komponente

4.3.1. Komponenta data_randomizing

Komponentu *data_ randomizing* čini Feistel struktura sa 18 rundi sa dva FL/FL^{-1} sloja, jedan nakon 6. runde i drugi nakon 12. runde. Ulazni parametri ove komponente su 128-bitni tekst nad kojim se vrši enkripcija označen kao *data_in* i 26 64-bitnih ključeva dobijenih iz *key_scheduling* komponente koji se koriste u rundama. Izlazni parametar je 128-bitni šifrovan tekst označen kao *data_out*.

Ulaz *data_in* se deli na dve celine od 64 bita, prvih 64 bita biće označeni sa *d1*, a drugih 64 bita sa *d2*.

Pre prve runde vrši se *pre-whitening*, nad parametrom *d1* i ključem *kw1* izvršava se XOR operacija, a takođe i nad parametrom *d2* i ključem *kw2*. Kod koji vrši ovu funkciju je sledeći:

```
int1 <= d1 xor kljucevi(22);
int2 <= d2 xor kljucevi(23);
```

Formiranje šifrovanog teksta podrazumeva kreiranje tri instance komponente "šest rundi" i dve instance komponente "FL" uz pravilno ulančavanje dotičnih instanci. Kreiran je novi tip *niz_registara* radi efikasnijeg unosa ključa u svakoj rundi.

```
type niz_registara isarray(0to25)ofstd_logic_vector(0to63);
signal kljucevi: niz_registara;
```

Kod koji vrši povezivanje rundi kao i unos ključa za svaku rundu je sledeći:

```
sest_rundi_inst1: sest_rundi
portmap(
        d1_in  => int1,
        d2_in => int2,
        k1 => kljucevi(0),
        k2 => kljucevi(1),
        k3 => kljucevi(2),
        k4 => kljucevi(3),
        k5 => kljucevi(4),
        k6 => kljucevi(5),
        d1_out  => int3,
        d2_out  => int4
        );

FL1:FL
portmap(
        fl_in =>int3,
        fl_k => kljucevi(6),
        fl_out => int5,
        fli_in =>int4,
        fli_k => kljucevi(7),
        fli_out => int6
        );

sest_rundi_inst2: sest_rundi
```

```
portmap(
        d1_in  => int5,
        d2_in => int6,
        k1 => kljucevi(8),
        k2 => kljucevi(9),
        k3 => kljucevi(10),
        k4 => kljucevi(11),
        k5 => kljucevi(12),
        k6 => kljucevi(13),
        d1_out  => int7,
        d2_out  => int8
        );

FL2:FL
portmap(
        fl_in => int7,
        fl_k => kljucevi(14),
        fl_out => int9,
        fli_in => int8,
        fli_k => kljucevi(15),
        fli_out => int10
        );

sest_rundi_inst3: sest_rundi
portmap(
        d1_in  => int9,
        d2_in => int10,
        k1 => kljucevi(16),
        k2 => kljucevi(17),
        k3 => kljucevi(18),
        k4 => kljucevi(19),
        k5 => kljucevi(20),
        k6 => kljucevi(21),
        d1_out  => int11,
        d2_out  => int12
        );
```

Nakon poslednje runde vrši se *post-whitening*, nad parametrom *int12* i ključem *kw3* izvšava se XOR operacija, a takođe i nad parametrom *int11* i ključem *kw4*. Kod koji vrši ovu funkciju je sledeći:

```
int13 <= int12 xor kljucevi(24);
int14 <= int11 xor kljucevi(25);
```

Konkatanacijom poslednja dva dobijena signala dobijamo izlaz, tačnije šifrovanu poruku:

```
data_out <= int13&int14;
```

Komponenta “šest rundi” koju smo detaljno opisali u prethodnom poglavlju formirana je ulančavanjem šest instanci “F” komponente. Kod kojim je realizovan ovaj segment je sledeći:

```
runda1: F
portmap(
        f_in  => d1_in,
```

```
        ke => kljucevi(0),
        f_out => int1
        );

runda2: F
portmap(
        f_in  =>d21 ,
        ke => kljucevi(1),
        f_out => int2
        );

runda3: F
portmap(
        f_in  => d11,
        ke => kljucevi(2),
        f_out => int3
        );

runda4: F
portmap(
        f_in  => d22,
        ke => kljucevi(3),
        f_out => int4
        );

runda5: F
portmap(
        f_in  => d12,
        ke => kljucevi(4),
        f_out => int5
        );

runda6: F
portmap(
        f_in  => d23,
        ke => kljucevi(5),
        f_out => int6
        );
```

4.3.2. Komponenta key_scheduling

Komponenta *key_scheduling* formira ključeve koji se koriste u okviru *data-randomizing* komponente za formiranje šifrovanog teksta.

Dakle, ulazni parametar ove komponente je 128-bitni ključ *k*, a izlazni parametri su 26 ključeva dužine 64 bita.

U okviru ove komponente instanciramo *generisanje_ka* komponentu, koja formira ključ *ka* na osnovu kog se dalje formiraju ključevi. Detaljna struktura ove komponente data je u prethodnom poglavlju.

U kodu kojim je realizovana ova komponenta instancirane su 4 komponente F-funkcije na sledeći način:

```
kl <= k;
kr <=(others=> '0');
d <= kl xor kr;
d1 <= d(0to63);
d2 <= d(64to127);

F1: F
portmap(
        f_in  => d1,
        ke => sigma1,
        f_out => int1
        );

d21 <= d2 xor int1;

F2: F
portmap(
        f_in  => d21,
        ke => sigma2,
        f_out => int2
        );

d11 <= d1 xor int2;

y <= d11&d21;
z <= y xor kl;
z1 <= z(0to63);
z2 <= z(64to127);

F3: F
portmap(
        f_in  => z1,
        ke => sigma3,
        f_out => int3
        );

z21 <= z2 xor int3;

F4: F
portmap(
        f_in  => z21,
        ke => sigma4,
        f_out => int4
        );

z11 <= z1 xor int4;

ka <= z11&z21;
```

Nakon što je generisan ključ *ka*, pristupamo formiranju ključeva koji se dobijaju cikličnim pomeranjem u levo za 15 ili 17 bita promenljivih *ka* i *kl*. Kod koji to izvršava je sledeći:

```
generisanje_ka_int: generisanje_ka
portmap(
        k  => k_int,
        ka => ka_int
        );
```

```
kljucevi(0)<= k_int(0to63);
kljucevi(1)<= k_int(64to127);

kljucevi(2)<= ka_int(0to63);
kljucevi(3)<= ka_int(64to127);

int1 <= k_int(15to127)&k_int(0to14);

kljucevi(4)<= int1(0to63);
kljucevi(5)<= int1(64to127);

int2 <= ka_int(15to127)&ka_int(0to14);

kljucevi(6)<= int2(0to63);
kljucevi(7)<= int2(64to127);

int3 <= ka_int(30to127)&ka_int(0to29);

kljucevi(8)<= int3(0to63);
kljucevi(9)<= int3(64to127);

int4 <= k_int(45to127)&k_int(0to44);

kljucevi(10)<= int4(0to63);
kljucevi(11)<= int4(64to127);

int5 <=ka_int(45to127)&ka_int(0to44);

kljucevi(12)<= int5(0to63);

int6 <= k_int(60to127)&k_int(0to59);

kljucevi(13)<= int6(64to127);

int7 <= ka_int(60to127)&ka_int(0to59);

kljucevi(14)<= int7(0to63);
kljucevi(15)<= int7(64to127);

int8 <= k_int(77to127)&k_int(0to76);

kljucevi(16)<= int8(0to63);
kljucevi(17)<= int8(64to127);

int9 <= k_int(94to127)&k_int(0to93);

kljucevi(18)<= int9(0to63);
kljucevi(19)<= int9(64to127);

int10 <= ka_int(94to127)&ka_int(0to93);

kljucevi(20)<= int10(0to63);
kljucevi(21)<= int10(64to127);

int11 <= k_int(111to127)&k_int(0to110);

kljucevi(22)<= int11(0to63);
kljucevi(23)<= int11(64to127);
```

```
int12 <= ka_int(111to127)&ka_int(0to110);

kljucevi(24)<= int12(0to63);
kljucevi(25)<= int12(64to127);
```

4.3.3. Komponenta Camellia

Ovaj entitet predstavlja logiku za šifrovanje po Camellia principu u okviru koga su instancirane komponente *data_randomizing* i *key_scheduling*. U pitanju je kombinaciona logika.

```
architecture shema of camellia is

signal data_in_int:STD_LOGIC_VECTOR(0to127);
signal kljuc_int:STD_LOGIC_VECTOR(0to127);
signal data_out_int:STD_LOGIC_VECTOR(0to127);
type niz_registara isarray(0to25)ofstd_logic_vector(0to63);
signal kljucevi: niz_registara;

component key_scheduling is
        port(
             k:inSTD_LOGIC_VECTOR(0to127);
             kw1:outSTD_LOGIC_VECTOR(0to63);
             kw2:outSTD_LOGIC_VECTOR(0to63);
             kw3:outSTD_LOGIC_VECTOR(0to63);
             kw4:outSTD_LOGIC_VECTOR(0to63);
             k1:outSTD_LOGIC_VECTOR(0to63);
             k2:outSTD_LOGIC_VECTOR(0to63);
             k3:outSTD_LOGIC_VECTOR(0to63);
             k4:outSTD_LOGIC_VECTOR(0to63);
             k5:outSTD_LOGIC_VECTOR(0to63);
             k6:outSTD_LOGIC_VECTOR(0to63);
             k7:outSTD_LOGIC_VECTOR(0to63);
             k8:outSTD_LOGIC_VECTOR(0to63);
             k9:outSTD_LOGIC_VECTOR(0to63);
             k10:outSTD_LOGIC_VECTOR(0to63);
             k11:outSTD_LOGIC_VECTOR(0to63);
             k12:outSTD_LOGIC_VECTOR(0to63);
             k13:outSTD_LOGIC_VECTOR(0to63);
             k14:outSTD_LOGIC_VECTOR(0to63);
             k15:outSTD_LOGIC_VECTOR(0to63);
             k16:outSTD_LOGIC_VECTOR(0to63);
             k17:outSTD_LOGIC_VECTOR(0to63);
             k18:outSTD_LOGIC_VECTOR(0to63);
             ke1:outSTD_LOGIC_VECTOR(0to63);
             ke2:outSTD_LOGIC_VECTOR(0to63);
             ke3:outSTD_LOGIC_VECTOR(0to63);
             ke4:outSTD_LOGIC_VECTOR(0to63)
             );
endcomponent;

component data_randomizing is
        port(
             data_in:inSTD_LOGIC_VECTOR(0to127);
             k1:inSTD_LOGIC_VECTOR(0to63);
             k2:inSTD_LOGIC_VECTOR(0to63);
             k3:inSTD_LOGIC_VECTOR(0to63);
             k4:inSTD_LOGIC_VECTOR(0to63);
```

```
            k5:inSTD_LOGIC_VECTOR(0to63);
            k6:inSTD_LOGIC_VECTOR(0to63);
            k7:inSTD_LOGIC_VECTOR(0to63);
            k8:inSTD_LOGIC_VECTOR(0to63);
            k9:inSTD_LOGIC_VECTOR(0to63);
            k10:inSTD_LOGIC_VECTOR(0to63);
            k11:inSTD_LOGIC_VECTOR(0to63);
            k12:inSTD_LOGIC_VECTOR(0to63);
            k13:inSTD_LOGIC_VECTOR(0to63);
            k14:inSTD_LOGIC_VECTOR(0to63);
            k15:inSTD_LOGIC_VECTOR(0to63);
            k16:inSTD_LOGIC_VECTOR(0to63);
            k17:inSTD_LOGIC_VECTOR(0to63);
            k18:inSTD_LOGIC_VECTOR(0to63);
            kw1:inSTD_LOGIC_VECTOR(0to63);
            kw2:inSTD_LOGIC_VECTOR(0to63);
            kw3:inSTD_LOGIC_VECTOR(0to63);
            kw4:inSTD_LOGIC_VECTOR(0to63);
            ke1:inSTD_LOGIC_VECTOR(0to63);
            ke2:inSTD_LOGIC_VECTOR(0to63);
            ke3:inSTD_LOGIC_VECTOR(0to63);
            ke4:inSTD_LOGIC_VECTOR(0to63);
            data_out:outSTD_LOGIC_VECTOR(0to127)
            );
endcomponent;

begin

        data_in_int <= data_in ;
        kljuc_int <= kljuc;


        key_scheduling_int: key_scheduling
        portmap(
                k  => kljuc_int,
                kw1  => kljucevi(0),
                kw2  => kljucevi(1),
                k1  => kljucevi(2),
                k2  => kljucevi(3),
                k3  => kljucevi(4),
                k4  => kljucevi(5),
                k5  => kljucevi(6),
                k6  => kljucevi(7),
                ke1  => kljucevi(8),
                ke2  => kljucevi(9),
                k7  => kljucevi(10),
                k8  => kljucevi(11),
                k9  => kljucevi(12),
                k10  => kljucevi(13),
                k11  => kljucevi(14),
                k12  => kljucevi(15),
                ke3  => kljucevi(16),
                ke4  => kljucevi(17),
                k13  => kljucevi(18),
                k14  => kljucevi(19),
                k15  => kljucevi(20),
                k16  => kljucevi(21),
                k17  => kljucevi(22),
                k18  => kljucevi(23),
```

```
                kw3   => kljucevi(24),
                kw4   => kljucevi(25)
                );

        data_randomizing_int: data_randomizing
        portmap(
                data_in   => data_in_int,
                kw1   => kljucevi(0),
                kw2   => kljucevi(1),
                k1   => kljucevi(2),
                k2   => kljucevi(3),
                k3   => kljucevi(4),
                k4   => kljucevi(5),
                k5   => kljucevi(6),
                k6   => kljucevi(7),
                ke1   => kljucevi(8),
                ke2   => kljucevi(9),
                k7   => kljucevi(10),
                k8   => kljucevi(11),
                k9   => kljucevi(12),
                k10   => kljucevi(13),
                k11   => kljucevi(14),
                k12   => kljucevi(15),
                ke3   => kljucevi(16),
                ke4   => kljucevi(17),
                k13   => kljucevi(18),
                k14   => kljucevi(19),
                k15   => kljucevi(20),
                k16   => kljucevi(21),
                k17   => kljucevi(22),
                k18   => kljucevi(23),
                kw3   => kljucevi(24),
                kw4   => kljucevi(25),
                data_out => data_out_int
                );


        data_out   <= data_out_int ;

end shema;
```

4.3.4. Opis rada Top-Level entiteta

Nakon što su objašnjene sve komponente potrebne za implementaciju algoritma, biće opisan kod u arhitekturi top-level entiteta:

```
    architecture shema of clocked_camellia is
    signal data_in_int:STD_LOGIC_VECTOR (0 to 127);
    signal kljuc_int:STD_LOGIC_VECTOR (0 to 127);
    signal data_out_int:STD_LOGIC_VECTOR (0 to 127);
    component camellia is
        port(
             data_in: in  STD_LOGIC_VECTOR (0 to 127);
             kljuc : in  STD_LOGIC_VECTOR (0 to 127);
             data_out: out  STD_LOGIC_VECTOR (0 to 127)
             );
    end component;
```

```
begin
        camellia_int: camellia
        port map(
                data_in => data_in_int,
                kljuc => kljuc_int,
                data_out => data_out_int
                );


    PROCESS(clk)

    begin

    IF(clk'EVENT AND clk='1')THEN

        data_in_int <= data_in ;
        kljuc_int <= kljuc;
        data_out <= data_out_int ;


    END IF;

    END PROCESS;


end shema;
```

Proces koji se nalazi unutar ovog entiteta predstavlja sinhronizovanje ulaza i izlaza Camellia entiteta. Kod unutar procesa se izvršava svaki put kada je aktivna uzlazna ivica takta. Ovim se obezbeđuje da se otvoreni tekst, ključ za šifrovanje i rezultat šifrovanja dostavljaju na ulaz, odnosno izlaz entiteta, sinhrono na takt.

5. OPIS PERFORMANSI I VERIFIKACIJA DIZAJNA

5.1. Verifikacija dizajna

U ovom poglavlju biće opisana procedura verifikacije dizajna. Za verifikaciju je korišćena funkcionalna simulacija u okviru ISim simulatora. Da bi se izvršila simulacija rada dizajna neophodno je kreirati odgovarajući testbenč. Testbenč predstavlja simulacioni VHDL entitet, koji okružuje dizajn koji se testira i koji generiše odgovarajuće pobude na ulazima dizajna na osnovu kojih se generiše scenario kojim se proverava ispravnost dizajna.

Na ulaze dizajna doveli smo više signala koji trebaju da prođu kroz proces šifrovanja.

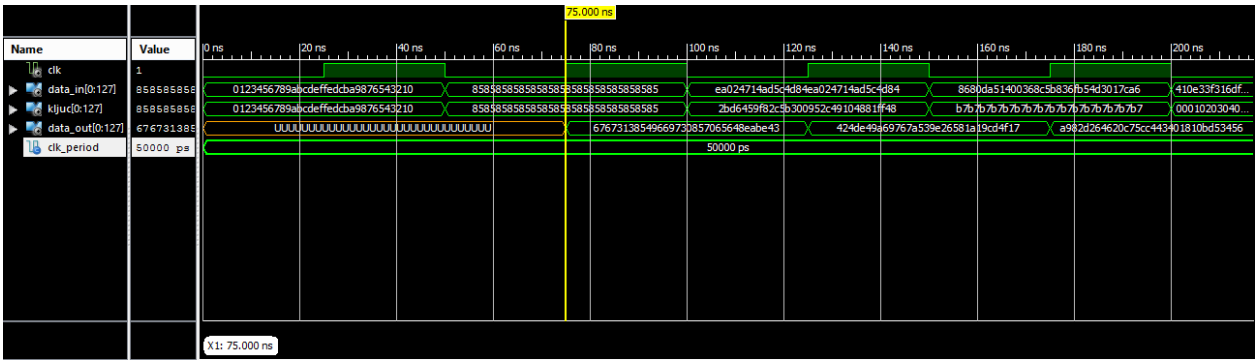
U tabeli 5.1.1. su prikazane vrednosti koje smo uzeli za proveru ispravnosti dizajna kao i rezultate koje treba da dobijemo nakon šifrovanja. Sve vrednosti su predstavljene u heksadecimalnom sistemu.

Tabela 5.1.1. Test vektori za korišćeni za verifikaciju dizajna

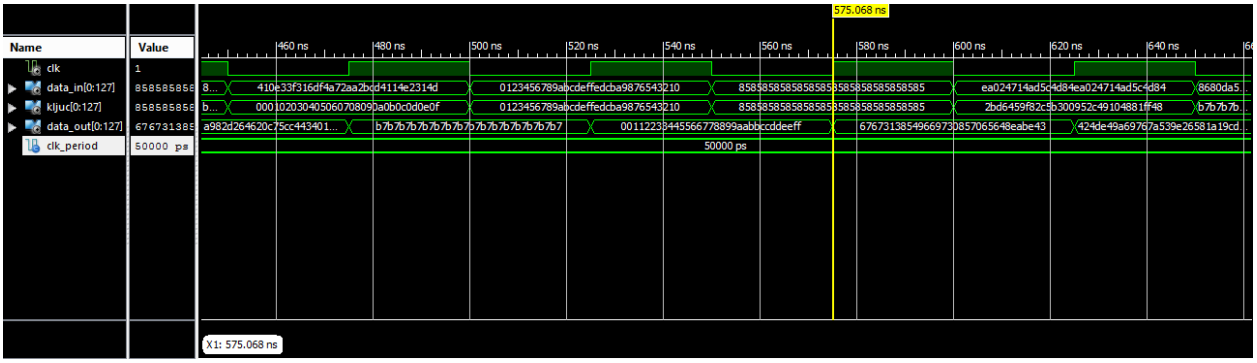
		Šifrovana poruka
Originalna poruka	0123456789ABCDEFFEDCBA9876543210	676731385496697308570656 48eabe43
Ključ	0123456789ABCDEFFEDCBA9876543210	
Originalna poruka	85858585858585858585858585858585	424DE49A69767A539E26581A19CD4F17
Ključ	85858585858585858585858585858585	
Originalna poruka	EA024714AD5C4D84EA024714AD5C4D84	A982D264620C75CC443401810BD53456
Ključ	2BD6459F82C5B300952C49104881FF48	
Originalna poruka	8680DA51400368C5B836FB54D3017CA6	B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7
Ključ	B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7	
Originalna poruka	410E33F316DF4A72AA2BCD4114E2314D	00112233445566778899AABBCCDDEEFF

Ključ	000102030405060708090A0B0C0D0E0F	

Na slikama 5.1.1. i 5.1.2. prikazani su rezultati simulacije. Vrednosti koje smo doveli na ulaze entiteta su vrednosti koje smo prikazali u prethodnoj tabeli. Kao rezultat pokretanja simulacije dobili smo šifrovane vrednosti koje smo i očekivali na izlazu entiteta.



Slika 5.1.1. Izgled prozora Isim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 1



Slika 5.1.2. Izgled prozora Isim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 2

5.2. Opis performansi

Proces analize i sinteze izvršen je u ISE razvojnom okruženju za FPGA čipove proizvođača Xilinx. Odabran je čip XC5VFX70T koji pripada Virtex 5 familiji čipova. Performanse dizajna prikazane su u tabeli 5.2.1.

Tabela 5.2.1. Performanse dizajna

	Korišćeno	Dostupno	Procenat iskorišćenosti
Broj slajs registara	384	44800	0%
Broj slajs LUT-ova	9994	44800	22%

Broj potpuno iskorišćenih parova LUT-FF	151	10227	1%
Broj pinova	385	640	60%
Broj globalnih taktova (BUFG/BUFGCTRL)	1	32	3%
Maksimalna frekvencija(MHz)	14.974 MHz		

Tabela prikazuje procenat zauzeća resursa. Iz prikazanih rezultata može se videti da dizajn ne troši mnogo hardverskih resursa.

Na osnovu maksimalne frekvencije odredili smo protok koji podržava naše rešenje i on iznosi 1.92 Gb/s. Protok bi mogao da se poveća korišćenjem pajplajn tehnike. Ova tehnika troši više resursa jer koristi registre kako bi ubrzala proces prosleđivanja obrađenih podataka, pošto se oni koriste za čuvanje međustanja. Pored registara ova tehnika zahteva i veći broj LUT-ova koji se koriste u kombinacionoj logici.

6. ZAKLJUČAK

U ovom radu prikazana je hardverska implementacija Camellia algoritma za enkripciju. Pri kreiranju ovog algoritma mislilo se i na napredak u metodama kriptoanalize, pa je dizajniran kako bi bio otporan ne samo na trenutne već i na neke buduće metode. Camellia algoritam naročito je pogodan za autentifikaciju i za bezbednu komunikaciju. Moguće je porediti ga sa AES standardom u pogledu nivoa sigurnosti i mogućnosti procesiranja.

Ovaj algoritam pored toga što je pogodan kako za softversku, tako i hardversku implementaciju, pruža i visok nivo zaštite. Sa praktične tačke gledišta algoritam je dizajniran tako da obezbedi fleksibilnu softversku i hardversku implementaciju na 32-bitnim procesorima korišćenim na Internetu, i mnoge aplikacije, kao i za 8-bitne procesore koji se koriste u pametnim karticama, integrisanim sistemima i drugim. Hardverska implementacija obezbeđuje veću sigurnost kao i veću brzinu obrade podataka. Realizovana implementacija ne koristi posebne resurse čipa poput interne memorije i sličnih, pa je stoga ovaj dizajn portabilan i bez izmena se može kompajlirati i za druge čipove istog, ali i drugih proizvođača.

Dati dizajn može se unaprediti korišćenjem pajplajn tehnike u procesu šifrovanja. Ova tehnika koristi registre između rundi za skladištenje trenutnog izlaza runde koja se izvršava. Kako se trenutne vrednosti rundi smeštaju u registar, ulaz za narednu rundu se uzima iz registra izbegavajući direktnu vezu između dve runde što obezbeđuje kontinualnu obradu podataka bez čekanja da se trenutni proces izvrši. Na ovaj način bi se omogućila veća brzina obrade podataka.

LITERATURA

[1] A.S. Tanenbaum, *Computer Networks*, Pearson, 2010.

[2] M. Matsui, J. Nakajima, A description of the Camellia Encryption Algorithm [Online] . Preuzeto sa : http://tools.ietf.org/html/rfc3713

[3] D. Denning, J. Irvine, M. Devlin, „A Key Agile 17.4 Gbit/sec Camellia Implementation“, *Field Programmable Logic and Application: 14th International Conference*, 2004

[4] K. Aoki, T. Ichikava, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, T. Tokita, Specification of Camellia – a 128-bit Block Cipher [Online] . Preuzeto sa : https://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf