SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINARSKI RAD

"CYFER" BIBLIOTEKA KRIPTOGRAFSKIH FUNKCIJA

Senko Rašić

Mentor: prof. dr. sc. Leo Budin

Zagreb, rujan 2004.

Sadržaj

1. Uvod

Cyfer je biblioteka kriptografskih funkcija, a sastoji se od nekoliko dijelova koji pokrivaju razna područja kriptografije:

- funkcije za izračunavanje sažetka poruke (*hash*),
- simetrični algoritmi za kriptiranje (*block, stream ciphers*),
- asimetrični algoritmi za kriptiranje (*public-key algorithms*),
- razmjena sjedničkih ključeva (*key-exchange algorithms*).

Za svako od ovih područja tijekom vremena razvijen je velik broj različitih algoritama, od kojih su neki praktično standard, a drugi se koriste ili u posebnim situacijama ili su potpuno zastarjeli i uopće se više ne koriste.

Cyfer biblioteka pruža ostvarenja relativno velikog broja kriptografskih algoritama, kako onih koji se sada koriste, tako i nekih koji su uključeni samo iz povijesnih razloga. Opis ostvarenih algoritama naveden je u 7. poglavlju ovog rada.

Kako bi se omogućila jednostavna nadogradnja i proširenje funkcionalnosti, biblioteka je organizirana modularno. Svako ostvarenje algoritma posebni je modul koji nema suvišne međuovisnosti s drugim dijelovima biblioteke. Takav modul moguće je i jednostavno fizički izdvojiti iz biblioteke i uključiti je u vanjskom programu, bez potrebe da se vuče i ostatak biblioteke. Ova mogućnost pogodna je za uporabu u ugrađenim (*embedded*) sustavima, kod kojih je potrebno minimizirati zauzeće memorije.

Pri izradi biblioteke obraćena je posebna pozornost prenosivosti na različite računalne platforme, što uključuje različite operacijske sustave (Unixoidi, MS Windows) i sklopovske platforme (Intel x86, Sun SPARC). Zavisnost o operacijskom sustavu svedena je na minimum, a u obzir su uzeta i svojstva sklopovlja na kojem se biblioteka koristi (podržani su i *little-endian* (x86) i *big-endian* (SPARC) sustavi, te veličine riječi od 32 i 64 bita).

Za izradu biblioteke korišten je programski jezik C, iako bi zbog objektno-orjentiranog dizajna prirodniji izbor bio C++. No, C++ pati od loše binarne uskladivosti sa ostalim jezicima, odnosno biblioteka pisana u C++-u se teško ili nikako ne može uključiti u programe pisane u drugim programskim jezicima. S druge strane, većina jezika podržava poziv C funkcija, pa je primjenjivost takve C biblioteke puno veća.

Osim samog ostvarenja u C-u, sa bibliotekom dolaze i omotači (*wrappers*) za popularne programske jezike koji omogućuju jednostavno i prirodno korištenje Cyfera iz tih jezika. Njihov API prilagođen je svakom jeziku posebno pa se ponešto razlikuje od C API-ja biblioteke, a dokumentacija je dostupna u elektroničnom obliku uz instalaciju pojedinih omotača.

2. Mogućnosti Cyfer biblioteke

Podržani algoritmi:

- funkcije za izračunavanje sažetka poruke: Adler-32, CRC-32, MD2, MD4, MD5, RIPEMD-160, SHA-1, SHA-256, Snefru,
- simetrični algoritmi: AES, Blowfish, DEAL, DES, DESX, TripleDES, IDEA, RC2, RC4, RC5, RC6, ThreeWay,
- načini rada simetričnih blok algoritama: ECB, CBC, CFB, OFB,
- asimetrični algoritmi: ElGamal, LUC, RSA,
- razmjena sjedničkih ključeva: Diffie-Hellman.

Podržani programski jezici (testirano na):

- C, C++ (GNU C Compiler, MS Visual Studio 7.1),
- Java (Sun Java2 SDK 1.4.2),
- C# (Mono 1.0, MS Visual Studio 7.1) – podrška za sve .NET jezike,
- Python (Python 2.2),
- Perl (Perl v5.8.0),
- PHP (PHP 4.3.1).

Testirane platforme:

- Linux, FreeBSD / x86,
- Solaris / SPARC v9,
- Windows XP.

Osim na testiranim platformama, Cyfer biblioteka trebala bi bez problema (ili uz minimalno izmjena) raditi na svim POSIX uskladivim operacijskim sustavima, uz uvjet da je na njima instalirano:

- *gmp* – GNU MP (multiple precision) biblioteka,
- *gcc* – GNU C prevodioc,
- *gmake* – GNU make.

3. Općeniti pregled biblioteke

Biblioteka je organizirana u tri dijela:

- *hash* (podrška funkcijama za izračunavanje sažetka poruke)
- *sym* (simetrični algoritmi; podrška za nekoliko načina rada)
- *asym* (asimetrični algoritmi; podrška za kriptiranje, digitalne potpise te razmjenu ključeva)

Svaki o navedenih dijelova podržava nekoliko najpopularnijih i najčešće korištenih algoritama za dotičnu primjenu. Prije korištenja samog algoritma, prvo je potrebno odabrati sam algoritam koji se želi koristiti. Svaki podržani algoritam poznat je biblioteci pod određenim imenom, kojeg je potrebno znati kako bi se on koristio. Korisniku (programeru) je dostupna i lista podržanih algoritama, sa bitnim informacijama o svojstvima svakog (duljina ključa, duljina bloka).

Nakon odabira algoritma, stvara se kontekst algoritma, koji sadrži sve podatke, privremene varijable, i sve drugo što je potrebno za rad pojedinog algoritma. Taj kontekst dalje se prenosi kao parametar ostalim funkcijama koje obavljaju sam posao, a na kraju posla se kontekst uništava.

Sami parametri koji se specificiraju prilikom stvaranja konteksta, te načini pozivanja funkcija, iako slični, razlikuju se u svakom od tri dijela biblioteke pa su zasebno opisani u narednim poglavljima.

Prilikom navođenja imena funkcija u daljnjem tekstu korištena je konvencija u kojoj se ne navodi prefiks `CYFER_` ispred imena funkcije, iako je on prisutan (u imenima svih funkcija, tipova podataka, konstanti i predefiniranih varijabli). Primjerice, ako se u tekstu spominje funkcija `Hash_Final()`, u kodu je potrebno pisati `CYFER_Hash_Final()`. U primjerima koda korištena su puna imena funkcija.

4. Funkcije za izračunavanje sažetka poruke

Hash funkcije, odnosno funkcije za izračunavanje sažetka poruke, koriste se kako bi se od proizvoljno velike poruke (odnosno, bilo kakvog podatkovnog objekta – pojam "poruka" može u ovom kontekstu zavaravati) izračunao sažetak točno određene duljine. Važno svojstvo funkcije za izračunavanje sažetka poruke jest ireverzibilnost, odnosno nemogućnost stvaranja (*preimage attack*) ili pronalaženja bilo koje poruke koja ima jednaku vrijednost sažetka kao neka određena poruka (*collision attack*). Kvaliteta, sigurnost i primjenjivost *hash* algoritma u kriptografske svrhe zasniva se upravo na njegovoj otpornosti na takve napade.

Biblioteka Cyfer podržava nekoliko najpopularnijih i najpoznatijih funkcija za izračunavanje sažetka poruke. Odabir funkcije za izračunavanje sažetka moguće je napraviti prilikom izrade programa koji koristi biblioteku (ugraditi odabir u izvornom kodu programa), ili to učiniti prilikom izvršavanja programa. Druga metoda omogućuje programu da na jednostavan i kontroliran način dozvoli korisniku programa da odabere koju funkciju želi koristiti.

Prilikom izvođenja programa moguće je saznati koji su podržani algoritmi pomoću funkcije `Hash_Get_Supported()`, koja vraća pokazivač na niz struktura `Hash_t` koje sadrže tipove, imena i duljine izlaza podržanih algoritama. Ako je unaprijed poznato koji se algoritam želi koristiti nije potrebno ručno prolaziti ovom listom, nego jednostavno pozvati funkciju `Hash_Select()`. Funkcija vraća tip algoritma i duljinu izlaza, ili javlja grešku ako algoritam nije podržan. Tip algoritma je broj koji se predaje funkciji `Hash_Init()` koja obavlja stvaranje i inicijalizaciju konteksta.

Nakon inicijalizacije, koristi se funkcija `Hash_Update()` za obradu podataka za koje je potrebno izračunati sažetak. Iako sami kriptografski algoritmi rade na blokovima određene duljine, to je skriveno od programera pa je moguće koristiti "blokove" podataka duljine od 0 pa do beskonačno (uvjetovano količinom memorije) okteta.

Na kraju rada, potrebno je pozvati funkciju `Hash_Final()` koja obavlja preostale korake algoritma, zatvara kontekst i vraća rezultat. U slučajevima kad je ulazni podatak dovoljno malen pa ga je moguće odjednom obraditi, moguće je koristiti pomoćnu funkciju `Hash()` koja objedinjuje otvaranje konteksta, računanje sažetka, te zatvaranje konteksta i vraćanje rezultata.

```
int md5sum(char *ulaz, size_t duljina, char *izlaz, size_t iz_duljina)
{
       size_t len;
       int type;
       CYFER_HASH_CTX *ctx;

       type = CYFER_Hash_Select("MD5", &len);
       if (type == CYFER_HASH_NONE) return -1;
       if (len > iz_duljina) return -1;

       ctx = CYFER_Hash_Init(type);
       if (!ctx) return -1;

       CYFER_Hash_Update(ctx, ulaz, duljina);
       CYFER_Hash_Final(ctx, izlaz);

       return 0;
}
```

Primjer 1: Računanje sažetka poruke

5. Simetrični kriptografski algoritmi

Simetrični kriptografski algoritmi koriste se za zaštitu tajnosti podataka – podatak (jasni tekst – *plaintext*) se kriptira (*encipherment*) tajnim ključem (*key, secret key*) koristeći neki algoritam (*cipher*). Izlaz tog algoritma (*ciphertext*) je takav da vanjskom napadaču ne daje nikakve informacije o jasnom tekstu, a poruku je moguće dekriptirati (*decipher*) samo uz posjedovanje tajnog ključa. Pošto se isti ključ koristi i za kriptiranje i za dekriptiranje poruka (podataka), cijeli sustav se naziva simetričnim kriptosustavom.

Danas je u širokoj uporabi velik broj simetričnih kriptografskih algoritama. Po načinu rada moguće ih je podijeliti u dvije skupine: algoritmi koji rade nad blokovima podataka (*block cipher*), te algoritmi koji rade na toku podataka (*stream cipher*). Algoritmi koji rade nad blokovima podataka podržavaju određene veličine bloka podataka, a do korisnika je da definira kako se kriptiraju nepotpuni blokovi. Algoritmi koji rade nad tokovima podataka temelje se na generatoru "slučajnih" (koji ovise o korištenom tajnom ključu) brojeva koji se kombiniraju sa ulazom kako bi se dobio kriptirani tekst. Danas se u najvećem broju slučajeva koriste blok-kriptografski algoritmi.

Biblioteka Cyfer podržava rad sa blok-kriptografskim algoritmima, kao i rad sa algoritmima koji rade nad tokom podataka. Kod blok algoritama, moguće je koristiti i neki od standardnih načina rada s blokovima: ECB (Electronic Cookbook Mode), CBC (Cipher Block Chaining), CFB (Cipher FeedBack) i OFB (Output FeedBack).

Slično kao kod funkcija za izračunavanje sažetka poruke, i kod simetričnih algoritama moguće je pozivom funkcije `BlockCipher_Get_Supported()` dobiti popis podržanih algoritama, duljina blokova nad kojima rade te duljina ključeva. Osim toga, Dostupna je i `BlockCipher_Get_SupportedModes()` funkcija kojom se dobija popis podržanih načina rada algoritma s duljinama blokova nad kojima rade, ako su različite od samog kriptografskog algoritma.

Funkcijama `BlockCipher_Select()` i `BlockCipher_SelectMode()` moguće je izravno odabrati koji se algoritam i način rada želi koristiti (odnosno provjeriti je li taj odabir podržan u biblioteci), a stvaranje i inicijalizacija konteksta obavlja se funkcijom `BlockCipher_Init()`. Prilikom inicijalizacije, potrebno je navesti i željenu duljinu ključa (ta vrijednost se zanemaruje kod algoritama koji ne podržavaju promjenjivu duljinu ključa) te, za neke načine rada, inicijalizacijski vektor (blok veličine bloka podataka sa nekim početnim vrijednostima).

Nakon postavljanja konteksta moguće je obavljati operacije enkripcije (`BlockCipher_Encrypt()`) ili dekripcije (`BlockCipher_Decrypt()`) nad blokovima podataka. Po završetku, potrebno je finalizirati i osloboditi kontekst, za što služe `BlockCipher_Finish()` i `BlockCipher_Delete()` funkcije.

```
int blowfish_encrypt(FILE *ulaz, FILE *izlaz, char *lozinka)
{
	size_t len, keylen, mlen;
	int type, mode;
	char *kljuc, txtulaz, txtizlaz;
	CYFER_BLOCK_CIPHER_CTX *ctx;

	type = CYFER_BlockCipher_Select("Blowfish", &len, &keylen);
	if (type == CYFER_CIPHER_NONE) return -1;

	mode = CYFER_BlockCipher_SelectMode("ECB", &mlen);
	if (mode == CYFER_MODE_NONE) return -1;
	if (mlen) len = mlen;

	// ako je kljuc duzi od lozinke, ostatak treba popuniti nulama
	kljuc = alloca(keylen);
	memset(kljuc, 0, keylen);
	strncpy(kljuc, lozinka, strlen(lozinka));

	// privremeni spremnici za jasni tekst i kriptirani tekst
	txtulaz = alloca(len);
	txtizlaz = alloca(len);

	// ECB ne koristi inicijalizacijski vektor ;-)
	ctx = CYFER_BlockCipher_Init(type, kljuc, keylen, mode, NULL);
	while (1) {
		int i = fread(txtulaz, 1, len, ulaz);
		if (i < 1) break;
		CYFER_BlockCipher_Encrypt(ctx, txtulaz, txtizlaz);
		fwrite(txtizlaz, 1, len, izlaz);
	}
	CYFER_BlockCipher_Finish(ctx);
	return 0;
}
```

Primjer 2: Enkripcija simetričnim blok algoritmom

Kod simetričnih algoritama koji rade nad tokovima podataka postupak je sličan ovome, osim što kod njih ne postoji definirana duljina bloka, pa se funkcijama za kriptiranje i dekriptiranje može predati proizvoljno velik blok podataka. Također, ne postoje različiti načini rada algoritma kao kod blok-orjentiranih algoritama.

Zbog toga postoje malene razlike u pozivima odgovarajućih `StreamCipher_*` funkcija – ne određuje se duljina bloka niti način rada, a kod svake enkripcije i dekripcije može se koristiti blok bilo koje duljine.

6. Asimetrični kriptografski algoritmi

Najveći problem kod korištenja simetričnih kriptografskih algoritama jest raspodjela ključeva. Kako obje strane u komunikaciji trebaju imati ključ, on se mora nekako raspodjeliti koristeći neki sigurni kanal – ali ako već postoji siguran kanal za komunikaciju, čemu uopće kriptirati poruke?

Ovaj problem kokoši i jajeta elegantno se rješava u asimetričnim kriptografskim algoritmima (*asymmetric ciphers, public-key cryptography*) korištenjem posebnog ključa za enkripciju a posebnog za dekripciju poruke. Svaki sudionik u raspravi stvara svoj par ključeva: javni (*public key*) i privatni (*private key*). Javni ključ služi za enkripciju podataka, a može se slobodno objaviti svima – jednom kriptirana poruka može se dekriptirati samo odgovarajućim privatnim ključem kojeg posjeduje samo onaj kojemu je poruka i namjenjena.

Druga popularna primjena asimetričnih algoritama je u elektroničnom potpisivanju poruka. U ovom slučaju poruka se kriptira privatnim ključem, a može ju dekriptirati bilo tko tko posjeduje javni ključ. Kako privatni ključ posjeduje samo jedna osoba, time se dokazuje identitet pošiljatelja poruke.

Asimetrični kriptografski algoritmi zasnivaju se "jednosmjernim funkcijama s tajnim vratima" (nespretan prijevod pojma *one-way trapdoor function*), odnosno matematičkim funkcijama kojima se inverz može lako izračunati samo uz poznavanje određene dodatne informacije (koja se koristi kao privatni ključ), a inače je njegovo računanje vrlo teško ili nemoguće. Najkorišteniji asimetrični algoritmi temelje se na problemu faktoriziranja velikih brojeva koji su umnožak dva prosta broja.

Cyfer podržava asimetrične algoritme i protokole za razmjenu sjedničkih ključeva, generiranje parova javni/tajni ključ, te za kriptiranje i digitalno potpisivanje poruka.

Za rad s velikim brojevima koji se koriste u tim algoritmima, Cyfer se oslanja na vanjsku biblioteku sa potrebnim rutinama. U svrhu minimiziranja ovisnosti o specifičnom ostvarenju, za sve operacije s velikim brojevima koriste se virtualne funkcije koje predstavljaju omotač oko API-ja za podršku velikih brojeva i mogu se jednostavno preraditi za korištenje raznih MP (multiple precision) biblioteka. Cyfer koristi GNU MP biblioteku.

Za razliku od računanja sažetka poruke i simetričnih algoritama, kod kojih su ključevi i podaci bilo kakvi nizovi okteta, u asimetričnoj kriptografiji pojavljuje se problem spremanja i interpretiranja podataka, odnosno pretvaranja ulaznih okteta u velike brojeve, kao i spremanje rezultata kao okteta. Tako postoje razni složeni standardi kodiranja zapisa i formata podataka, od kojih su najpoznatiji PKCS standardi.

Cyfer izbjegava cijeli problem tako što se uopće ne brine za ulaz i izlaz podataka. Svi podaci su čisti nizovi okteta s kojima korisnik (programer) može raditi što god želi, uz poštivanje ograničenja na veličinu tih podataka. Na taj način programeru je dana sloboda za bilo kakvo ostvarenje bez nametanja određenog standarda.

Podrška za asimetrične algoritme u Cyfer-u sastoji se od podrške za generiranje ključeva, kriptiranje i dekriptiranje blokova podataka (određene duljine), kao i potpisivanje poruka i provjeru potpisa nad blokovima podataka (određene duljine). Podrška za potpise zapravo predstavlja inverzne operacije onima za kriptiranje podataka.

Kao i kod ostatka Cyfer biblioteke, prije samog rada sa algoritmom potrebno je stvoriti i inicijalizirati kontekst algoritma, a po završetku rada finalizirati i osloboditi kontekst. Odabir algoritma koji se koristi obavlja se funkcijom `Pk_Select()`, koja osim vraćanja odabranog tipa algoritma postavlja i dvije zastavice, koje signaliziraju može li se odabranim algoritmom kriptirati i potpisivati poruke. Osim nje, postoji i `Pk_Get_Supported()` funkcija koja vraća popis podržanih algoritama (u obliku pokazivača na niz stuktura koje opisuju algoritam).

```
bool enc, sig;
int type = CYFER_Pk_Select("RSA", &enc, &sig);
CYFER_PK_CTX *ctx = CYFER_Pk_Init(type);
       ...
CYFER_Pk_Finish(ctx);
```

Primjer 3: Kostur programa koji koristi asimetrične algoritme

6.1. Generiranje ključeva

Svi asimetrični algoritmi ostvareni u Cyfer biblioteci podržavaju ključeve proizvoljne duljine. Generiranje para javni/privatni ključ obavlja se pomoću funkcije `Pk_Generate_Key()` kojoj se kao parametar prenosi tražena duljina ključeva. Nakon generiranja para, moguće ga je spremiti u podatkovni spremnik kao niz okteta. Potrebne veličine spremnika mogu se dobiti pozivom funkcije `Pk_KeySize()`.

Izvoz i uvoz privatnog i javnog ključa obavlja se pomoću funkcija `Pk_Export_Key()` i `Pk_Import_Key()`. Iako obje funkcije mogu uvesti ili izvesti i privatni i javni ključ, moguće je za ključ koji je nepoznat ili se ne želi izvesti specificirati `NULL` vrijednost spremnika, čime se taj ključ ignorira.

```
size_t bits = 1024; // koristi 1024-bitne kljuceve
size_t pub_len, priv_len;
char *priv_key, *pub_key;

// generiraj kljuceve
CYFER_Pk_Generate_Key(ctx, bits);
// odredi koliko okteta treba za spremanje kljuceva
CYFER_Pk_KeySize(ctx, &priv_len, &pub_len);

priv_key = alloca(priv_len);
pub_key = alloca(pub_len);

// izvezi privatni i javni kljuc
CYFER_Pk_Export_Key(ctx, priv_key, pub_key);
       ...
// zanima nas samo javni kljuc
CYFER_Pk_Import_Key(ctx, NULL, 0, pub_key, pub_len);
```

Primjer 4: Generiranje, uvoz i izvoz ključeva

6.2. Kriptiranje i potpisivanje

Operacije kriptiranja, dekriptiranja, potpisivanja te provjere potpisa obavljaju se nad blokovima podataka. Pri tome treba obratiti pažnju na činjenicu da veličina bloka jasnog teksta i veličina bloka kriptiranog teksta ne moraju biti jednake.

Funkcija `Pk_Sign()` koja obavlja potpisivanje podatka identična je funkciji `Pk_Encrypt()`, samo što koristi privatni a ne javni ključ. S druge strane, `Pk_Verify()` se od `Pk_Decrypt()` razlikuje po tome što ne vraća originalnu poruku u spremniku, već uspoređuje originalnu poruku (koja mora biti proslijeđena u spremniku) sa kriptiranom porukom.

```
void kriptiraj(CYFER_PK_CTX *ctx, FILE *in, FILE *out)
{
	// velicina bloka: jasnog teksta, kriptiranog teksta
	size_t pt_len, ct_len;
	char *pt, *ct;

	// odredi velicine i alociraj spremnike
	CYFER_Pk_Size(ctx, &pt_len, &ct_len);

	pt = alloca(pt_len);
	ct = alloca(ct_len);

	while (1) {
		memset(pt, 0, pt_len);
		int n = fread(pt, 1, pt_len, in);
		if (n < 1) break;
		CYFER_Pk_Encrypt(ctx, pt, ct);
		fwrite(ct, 1, ct_len, out);
	}
}
```

Primjer 5: Kriptiranje datoteke asimetričnim algoritmom

6.3. Razmjena sjedničkih ključeva

Razmjenu sjedničkih ključeva moguće je ostvariti bilo kojim asimetričnim kriptografskim algoritmom, tako što svaka strana generira neki slučajni ključ, kriptira ga javnim ključem druge strane, i pošalje drugoj strani. No kako je razmjena sjedničkih ključeva vrlo česta u uporabi, razvili su se i posebni algoritmi koji omogućuju efikasnu razmjenu.

Cyfer trenutno podržava razmjenu ključeva Diffie-Hellman algoritmom, ali je zbog kasnije lakše nadogradnje ostvareno i generičko sučelje (API) za tu funkcionalnost. Sučelje je slično funkcijama za rad sa općenitim asimetričnim algoritmima, ali su neke stvari prilagođene specifičnostima problema razmjene ključeva.

Protokol razmjene sjedničkih ključeva odvija se u slijedećim koracima (svaka strana obavlja jednak posao):

1. stvara se i inicijalizira kriptografski kontekst,
2. stvara se privatni i javni ključ (`KeyEx_Generate_Key()`),
3. javni ključ se razmijeni sa drugom stranom (`KeyEx_Public_Key()`),
4. ma osnovu javnog ključa druge strane i vlastitog privatnog ključa računa se sjednički ključ (`KeyEx_Compute_Key()`),
5. sjednički ključ se izveze u blok podataka određene duljine i može se dalje koristiti za sigurnu komunikaciju (`KeyEx_Shared_Key()`).

```
int razmjena_kljuceva(char *sjednicki, int trazena_duljina)
{
	CYFER_KEYEX_CTX *ctx;
	size_t duljina_javni, duljina_drugastrana;
	int type;
	char *javni, *drugastrana;

	type = CYFER_KeyEx_Select("DH");
	if (type == CYFER_KEYEX_NONE) return -1;
	ctx = CYFER_KeyEx_Init(type);
	if (!ctx) return -1;

	// generiranje para kljuceva
	CYFER_KeyEx_Generate_Key(ctx);
	CYFER_KeyEx_KeySize(ctx, NULL, &duljina_javni);

	// izvoz javnog kljuca
	javni = alloca(duljina_javni);
	CYFER_KeyEx_Public_Key(ctx, javni);

	... // razmjena javnog kljuca s drugom stranom

	// racunanje sjednickog kljuca
	CYFER_KeyEx_Compute_Key(ctx, drugastrana, duljina_drugastrana);

	// mozemo zatraziti sjednicki kljuc bilo koje duljine
	// on se racuna iz generiranog sjednickog kljuca
	CYFER_KeyEx_Shared_Key(ctx, sjednicki, trazena_duljina);

	CYFER_KeyEx_Finish(ctx);
	return 0;
}
```

Primjer 6: Razmjena sjedničkih ključeva

7. Ostvareni kriptografski algoritmi

Ideja Cyfer biblioteke je pružiti što je moguće više različitih algoritama korisniku na izbor, no kako ih nije moguće (niti potrebno) ostvareni sve, prilikom izrade biblioteke trebalo je na neki način odrediti koje algoritme podržati a koje ne.

U izbor su ušli oni algoritmi koji imaju ili neposrednu praktičnu vrijednost (npr. SHA-1, RSA i AES su visokokvalitetni algoritmi i standard na području kriptografije), ili imaju veliko povjesno značenje i pružaju općenit uvid u dizajn kriptografskih algoritama (npr DES sa svojom revolucionarnom *Feistel* strukturom), ili su se zbog nekog drugog razloga činili prikladnim za uključivanje u biblioteku.

Prilikom proučavanja algoritama, njihovog ostvarenja i evaluacije korištene su bilješke s predavanja iz kolegija "Operacijski sustavi 2" na Fakultetu Elektrotehnike i Računarstva (FER) u Zagrebu ([1]), primjeri i dokumentacija iz studentskih projekata iz područja računalne sigurnosti na FER-u ([2]), knjiga "*Handbook of Applied Cryptography*" ([3]), službene specifikacije algoritama te razni članci s Interneta.

7.1. Korištene konvencije prilikom opisivanja algoritama

U opisu algoritma navedeno je nad kakvim podacima se obavljaju operacije. Neki algoritmi rade nad pojedinim oktetima, neki nad riječima od 2 znaka (16 bita), a neki nad riječima od 4 znaka (32 bita).

Kod algoritama koji koriste razne tablice pretraživanja (npr. za permutacije ili selekcije bitova) navedeno je samo ime tablice kako se koristi i u ostvarenju, bez ispisivanja samog sadržaja tablice koje ne bi pridonijelo razumijevajnu algoritma i samo bi nepotrebno zauzimalo prostor u dokumentaciji.

Korištene oznake:

- `X xor Y` – binarna operacija isključivo ILI
- `X or Y` – binarna operacija ILI
- `X and Y` – binarna operacija I
- `not(X)` – binarna operacija NE (jedinični komplement)
- `rotl(X,N)` – rotacija X u lijevo za N bitova
- `rotr(X,N)` – rotacija X u desno za N bitova
- `shl(X,N)` – pomak X u lijevo za N bitova
- `shr(X,N)` – pomak X u desno za N bitova
- `X mod Y` – ostatak dijeljenja X sa Y
- `(A, B, C) = (D, E, F)` – istovremeno dodjeljivanje vrijednosti (A = stari D, B = stari E, C = stari F)
- `X[a .. a+k] = Y[b .. b+k]` – dio niza Y kopira se u niz X

7.2. Funkcije za izračunavanje sažetka poruke

7.2.1. MD2

MD2 algoritam za računanje sažetka poruke radi nad ulaznim blokovima duljine 16 okteta, a kao rezultat daje sažetak poruke veličine također 16 okteta, odnosno 128 bita. Autor algoritma je RSA Data Security, a dozvoljeno ga je koristiti u nekomercijalne svrhe za Privacy Enhanched Mail (PEM) poruke na Internetu. Specifikacija algoritma dana je u [4].

Prije računanja sažetka poruke, potrebno ju je produžiti tako da njena veličina (u oktetima) bude višekratnik broja 16. Poruka se uvijek produžuje, čak i u slučaju da njena osnovna duljina već je višekratnik broja 16; u tom slučaju poruka se produžuje za 16 okteta. Poruka se dopunjava oktetima ASCII vrijednosti X, gdje je X broj okteta koji se dodaju poruci ($1 \leq X \leq 16$). Primjerice, ako je poruci potrebno dodati još 4 znaka, bit će joj dodani okteti (zapis u C string notaciji): "\x04\x04\x04\x04"; ako je potrebno dodati 2 znaka, bit će dodano "\x02\x02".

Nakon dopunjavanja poruke, dodaje joj se još jedan blok od 16 okteta, koji predstavlja dodatnu zaštitnu sumu poruke (*checksum*). U računanju zaštitne sume upotrebljava se i predefinirana tablica 256 "slučajnih" permutacija okteta, ovdje označena sa S. Zaštitna suma (C[i], $0 \leq i \leq 15$) računa se slijedećim algoritmom (M[i] označava i-ti oktet poruke, N je broj okteta dopunjene poruke, sve operacije obavljaju se nad jednim oktetom):

```
Za i = 0 do 15: C[i] = 0.
L = 0.

Za i = 0 do (N / 16) - 1:
       Za j = 0 do 15:
              L = C[j] = S[M[i * 16 + j] xor L].
       Kraj j.
Kraj i.
```

Ispis 1: Dopunjavanje poruke u MD2 algoritmu

Nakon što se izračunata zaštitna suma doda poruci (čija nova duljina je N' = N + 16) pokreće se drugi dio algoritma. Ovaj dio se sastoji od 18 koraka, a u njegovu radu upotrebljava se i 48-oktetni spremnik, ovdje označen kao X:

```
Za i = 0 do 48: X[i] = 0.

Za i = 0 do (N' / 16) - 1:
       Za j = 0 do 15:
              X[16 + j] = M[i * 16 + j].
              X[32 + j] = X[16 + j] xor X[j].
       Kraj j.
       t = 0.
       Za j = 0 do 17:
              Za k = 0 do 47:
                     t = X[k] = X[k] xor S[t].
              Kraj k.
              t = (t + j) mod 256.
       Kraj j.
Kraj i.
```

Ispis 2: Funkcija sažimanja MD2 algoritma

Po završetku drugog dijela algoritma vrijednosti X[i], $0 \leq i \leq 15$ koriste se kao rezultat cijelog MD2 algoritma, odnosno predstavljaju sažetak izvorne poruke.

7.2.2. MD4

MD4 algoritam je drugi u seriji algoritama za računanje sažetka poruke razvijenih od tvrtke RSA Data Security. Ovaj algoritam stavljen je u javno vlasništvo, a specifikacija i referentno ostvarenje mogu se naći u [5]. Algoritam je optimiran za izvođenje na 32-bitnim računalima. Rezultat algoritma (sažetak poruke) je veličine 128 bita (16 okteta).

Algoritam MD4 radi nad riječima duljine 32 bita (4 znaka). Pretvaranje niza okteta u riječi i obrnuto obavlja se prema *little-endian* notaciji, odnosno najmanje značajni oktet riječi je zapisan prvi.

Poruka se prije računanja sažetka dopunjava nizom bitova P i dodaje joj se 64-bitni broj L koji predstavlja duljinu poruke u bitovima. Nakon dopunjavanja duljina poruke M u bitovima mora biti djeljiva sa 512 (odnosno, poruka se mora moći razbiti u cijeli broj blokova od 64 znaka). Pošto je duljina broja L 64 bita, to znači da je potrebno poruku tako dopuniti da njena duljina u bitovima pri djeljenju sa 512 daje ostatak 448. Poruka se uvijek dopunjava, čak i ako je njena duljina već odgovarajuća, što znači da se poruci dodaje 1 do 512 bitova. Prilikom popunjavanja kao prvi bit koristi se 1, a ostali bitovi su 0.

U slučaju kad je poruka niz okteta, njena duljina u bitovima je sigurno višekratnik broja 8. Tada se dopunjavanje svodi na dodavanje 1 do 64 okteta, od kojih prvi ima vrijednost 0x80 (dekadski 128, postavljen najviši bit), a ostali su 0x00.

Nakon dopunjavanja bitova, poruci se dodaje 64-bitni broj L, također prema *little-endian* notaciji, tj. prvo se sprema njegova manje značajna riječ, a nakon nje značajnija riječ. Dobivena poruka M ima duljinu u bitovima koja je višekratnik broja 512. Svakih 512 bitova je moguće predstaviti kao 16 32-bitnih riječi, pa se poruka može zapisati kao niz 32-bitnih riječi M[0] .. M[N – 1], gdje je N višekratnik broja 16.

Prilikom rada algoritam koristi spremnik od četiri 32-bitne riječi (A, B, C, D) za računanje rezultata. Te riječi inicijalizirane su na vrijednosti:

```
A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476
```

Osim toga, definirane su i 3 pomoćne funkcije, za svaki od koraka po jedna:

```
f(X, Y, Z) = (X and Y) or (not(X) and Z)
g(X, Y, Z) = (X and Y) or (X and Z) or (Y and Z)
h(X, Y, Z) = X xor Y xor Z
```

Definirane su i 3 tablice transformacija (aditivne konstante y[i], rotacije s[i] i permutacije z [i]) koje sadrže po 48 elemenata. U nekim ostvarenjima algoritma ove transformacije definiraju se pomoćnim funkcijama (njih 48) koje se zapisuju izravno u kod, što zapravo predstavlja razmatanje petlje (*loop-unrolling*) i povećava učinkovist koda ali znatno smanjuje njegovu čitljivost.

Sa ovako definiranim konstantama i transformacijama, funkcija sažimanja MD4 algoritma glasi:

```
Za i = 0 do (N / 16) – 1:
       (AA, BB, CC, DD) = (A, B, C, D).
       X[0..15] = M[i * 16 .. i * 16 + 15].

       /* prvi korak */
       Za j = 0 do 15:
              t = A + f(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, rotl(t, s[j]), B, C).
       Kraj j.
       /* drugi korak */
       Za j = 16 do 31:
              t = A + g(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, rotl(t, s[j]), B, C).
       Kraj j.

       /* treci korak */
       Za j = 32 do 47:
              t = A + h(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, rotl(t, s[j]), B, C).
       Kraj j.

       (A, B, C, D) = (A + AA, B + BB, C + CC, D + DD).
Kraj i.
```

Ispis 3: Funkcija sažimanja MD4 algoritma

Izlaz iz algoritma (sažetak poruke) dobije se spajanjem riječi A, B, C i D.

7.2.3. MD5

Algoritam MD5 dizajniran je kao pojačana verzija MD4 već i prije nego li su se otkrile slabosti tog algoritma. Nešto je sporiji od MD4, ali ima veću razinu sigurnosti. Algoritam je razvijen u tvrtci RSA Data Security, stavljen je u javno vlasništvo i specificiran u [6], a u praksi je našao široku primjenu.

Algoritam je optimiran za izvođenje na 32-bitnim računalima, a izlaz mu je 128-bitni. Za razliku od MD4, algoritam koristi 4 koraka i ima malo drugačije definirane transformacije. Zbog sličnosti sa MD4, ovdje su opisane samo razlike u odnosu na taj algoritam.

Pomoćna funkcija u koraku 2 redefinirana je kao:

```
g(X, Y, Z) = (X and Z) or (Y and not(Z))
```

Nova pomoćna funkcija za korak 4 glasi:

```
k(X, Y, Z) = Y xor (X and not(Z))
```

Transformacijske tablice s[i], y[i] i z[i] izmjenjene su i proširene na 64 elementa. U svakom koraku promjenjeno je računanje registra B, a uvedena je i već spomenuti četvrti korak. Postupci pripreme poruke, inicijalizacije algoritma i vraćanja rezultata su nepromijenjeni. Sama funkcija sažimanja glasi:

```
Za i = 0 do (N / 16) - 1:
       (AA, BB, CC, DD) = (A, B, C, D).
       X[0..15] = M[i * 16 .. i * 16 + 15].

       /* prvi korak */
       Za j = 0 do 15:
              t = A + f(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, B + rotl(t, s[j]), B, C).
       Kraj j.
       /* drugi korak */
       Za j = 16 do 31:
              t = A + g(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, B + rotl(t, s[j]), B, C).
       Kraj j.

       /* treci korak */
       Za j = 32 do 47:
              t = A + h(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, B + rotl(t, s[j]), B, C).
       Kraj j.

       /* cetvrti korak */
       Za j = 48 do 63:
              t = A + k(B, C, D) + X[z[j]] + y[j].
              (A, B, C, D) = (D, B + rotl(t, s[j]), B, C).
       Kraj j.

       (A, B, C, D) = (A + AA, B + BB, C + CC, D + DD).
Kraj i.
```

Ispis 4: Funkcija sažimanja MD5 algoritma

7.2.4. SHA-1

Secure Hash Algorithm (SHA) je algoritam temeljen na MD4, predložen je od strane američkog nacionalnog instituta za standarde i tehnologiju (NIST – *National Institute for Standards and Technology*) za uporabu u državnim ustanovama i agencijama, a moguće ga jekoristiti u privatne i komercijalne svrhe.

Algoritam je objavljen u NIST publikaciji 180, a u njenoj drugoj reviziji [7] standardizirane su i proširene verzije algoritma koje daju izlaz širine 256, 384 i 512 bita. SHA-1 algoritam daje izlaz od 160 bitova, a radi nad blokovima od 512 bitova, koji se interpretiraju kao 16 32-bitnih riječi. Pretvorba iz okteta u riječi i obrnuto obavlja se po *big-endian* notaciji, dakle suprotno od MD4 i MD5 algoritama.

Prije ulaza u algoritam, poruka se dopunjava na način identičan onome kod MD4, osim što se duljina poruke zapisuje po *big-endian* notaciji (prvo značajnija riječ). U algoritmu se koristi 5 spremnika privremenih vrijednosti:

```
A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476
E = 0xc3d2e1f0
```

Definiraju se i 4 aditivne konstante:

```
y1 = 0x5a827999; y2 = 0x6ed9eba1; y3 = 0x8f1bbcdc; y4 = 0xca62c1d6
```

SHA-1 funkcija sažimanja je (pomoćne funkcije f, g, h su preuzete od MD4):

```
Za i = 0 do (N / 16) – 1:
       (AA, BB, CC, DD, EE) = (A, B, C, D, E).
       X[0..15] = M[i * 16 .. i * 16 + 15].
       Za j = 16 do 79:
              t = X[j – 3] xor X[j – 8] xor X[j – 14] xor X[j – 16].
              X[j] = rotl(t, 1).
       Kraj j.

       Za j = 0 do 20:
              t = rotl(A, 5) + f(B, C, D) + E + X[j] + y1.
              (A, B, C, D, E) = (t, A, rotl(B, 30), C, D).
       Kraj j.
       Za j = 0 do 20:
              t = rotl(A, 5) + g(B, C, D) + E + X[j] + y2.
              (A, B, C, D, E) = (t, A, rotl(B, 30), C, D).
       Kraj j.
       Za j = 0 do 20:
              t = rotl(A, 5) + h(B, C, D) + E + X[j] + y3.
              (A, B, C, D, E) = (t, A, rotl(B, 30), C, D).
       Kraj j.
       Za j = 0 do 20:
              t = rotl(A, 5) + g(B, C, D) + E + X[j] + y4.
              (A, B, C, D, E) = (t, A, rotl(B, 30), C, D).
       Kraj j.
       (A, B, C, D) = (A + AA, B + BB, C + CC, D + DD).
Kraj i.
```

Ispis 5: Funkcija sažimanja SHA-1 algoritma

Izlaz iz algoritma su spojene vrijednosti A, B, C, D i E, pretvorene u niz prema *big-endian* notaciji.

7.2.5. SHA-256

SHA-256 je proširenje algoritma SHA-1, definirano u [7]. SHA-256 radi nad blokovima od 512 bitova, koje se interpretiraju kao 16 32-bitnih riječi po *big-endian* notaciji. Algoritam koristi 64 konstanti K[i], 64 pomoćnih riječi W[i], 8 spremnika privremenih vrijednosti (a, b, c, d, e, f, g, h) i privremenu vrijednost sažetka H[i] od 8 riječi.

SHA-256 koristi nekoliko pomoćnih funkcija:

```
Ch(x, y, z) = (x and y) xor (not(x) and z)
Maj(x, y, z) = (x and y) xor (x and z) xor (y and z)
S0(x) = rotr(x, 2) xor rotr(x, 13) xor rotr(x, 22)
S1(x) = rotr(x, 6) xor rotr(x, 11) xor rotr(x, 25)
s0(x) = rotr(x, 7) xor rotr(x, 18) xor shr(x, 3)
s1(x) = rotr(x, 17) xor rotr(x, 19) xor shr(x, 10)
```

Privremena vrijednost sažetka se inicijalizira konstantama:

```
H[0] = 0x6a09e667
H[1] = 0xbb67ae85
H[2] = 0x3c6ef372
H[3] = 0xa54ff53a
H[4] = 0x510e527f
H[5] = 0x9b05688c
H[6] = 0x1f83d9ab
H[7] = 0x5be0cd19
```

Funkcija sažimanja SHA-256 algoritma je (zbrajanje se koristi kao modulo 2^{32} operacija):

```
Za i = 0 do (N / 16) – 1:
      W[0..15] = M[i * 16 .. i * 16 + 15].
      Za j = 16 do 63:
            W[j] = s1(W[j – 2]) + W[j – 7] +
                  s0(W[j – 15]) + W[j – 16].
      Kraj j.
      (a, b, c, d) = (H[0], H[1], H[2], H[3]).
      (e, f, g, h) = (H[3], H[4], H[5], H[6]).

      Za j = 0 do 63:
            t1 = h + S1(e) + Ch(e, f, g) + K[j] + W[j].
            t2 = S0(a) + Maj(a, b, c).
            (h, g, f, e) = (g, f, e, d + t1).
            (d, c, b, a) = (c, b, a, t1 + t2).
      Kraj j.
      H[0] = H[0] + a.
      H[1] = H[1] + b.
      H[2] = H[2] + c.
      H[3] = H[3] + d.
      H[4] = H[4] + e.
      H[5] = H[5] + f.
      H[6] = H[6] + g.
      H[7] = H[7] + h.
Kraj i.
```

Ispis 6: Funkcija sažimanja SHA-256 algoritma

Rezultat algoritma dobiva se iz varijabli H[i], pretvaranjem u niz prema *big-endian* notaciji.

7.2.6. RIPEMD-160

RIPEMD-160 je funkcija za računanje sažetka poruke nastala na osnovu iskustava u analizi MD4, MD5 i RIPEMD algoritama. Objavljena je 1996. godine u [8], a smatra se jednako sigurnom kao i SHA-1.

RIPEMD-160 algoritam radi nad ulaznim blokovima od 512 bita (16 32-bitnih riječi, konverzija se obavlja po *little-endian* notaciji), a daje rezultat veličine 160 bita. Izvorna poruka dopunjava se kao kod MD4. Algoritam koristi 5 spremnika privremenih vrijednosti, koji se inicijaliziraju na vrijednosti:

```
A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476
E = 0xc3d2e1f0
```

Koristi se 5 pomoćnih funkcija koje su definirane kao:

```
q[0] = f(u, v, w) = u xor v xor w
q[1] = g(u, v, w) = (u and v) or (not(u) and w)
q[2] = h(u, v, w) = (u or not(v) xor w)
q[3] = k(u, v, w) = (u and w) or (v and not(w))
q[4] = l(u, v, w) = u xor (v or not(w))
```

Algoritam se obavlja paralelno u dvije linije, nad lijevom i desnom polovicom ulaza. Koristi se 10 aditivnih konstanti (po 5 za svaku liniju), yL[i] i yR[i], $0 \leq i \leq 4$. Definirane su i tablice pomaka i permutacija za lijevu i desnu liniju, sL[i], sR[i], zL[i] i zR[i], $0 \leq i \leq 79$.

```
Za i = 0 do (N / 16) – 1:
      X[0..15] = M[i * 16 .. i * 16 + 15].
      (AL, BL, CL, DL, EL) = (A, B, C, D, E).
      (AR, BR, CR, DR, ER) = (A, B, C, D, E).

      Za j = 0 do 4:
            Za k = 0 do 15:
                  m = j * 16 + k.
                  t = AL + q[j](BL, CL, DL) + X[zL[m]] + yL[j].
                  (AL, BL, CL, DL, EL) = (EL, EL + rotl(t, sL[m]),
                        BL, rotl(CL, 10), DL).
            Kraj k.
      Kraj j.

      Za j = 0 do 4:
            Za k = 0 do 15:
                  m = j * 16 + k.
                  t = AR + q[4 - j](BR, CR, DR) + X[zR[m]] + yR[j].
                  (AR, BR, CR, DR, ER) = (ER, ER + rotl(t, sR[m]),
                        BR, rotl(CR, 10), DR).
            Kraj k.
      Kraj j.

      t = A.
      A = B + CL + DR.
      B = C + DL + ER.
      C = D + EL + AR.
      D = E + AL + BR.
      E = t + BL + CR.
Kraj i.
```

Ispis 7: Funkcija sažimanja RIPEMD-160 algoritma

7.2.7. Snefru

Funkciju za računanje sažetka poruke Snefru objavio je 1990. godine Ralph Merkle u [9]. Funkcija je parametrizirana veličinom izlaza, odnosno može raditi sa različitim duljinama ulaznih blokova i izlaznih vrijednosti. Ovdje je opisana verzija algoritma koja daje rezultat duljine 128 bita. Ta verzija koristi ulazne blokove od 384 bita (48 okteta).

Algoritam koristi *big-endian* notaciju za pretvaranje nizova okteta u riječi i obrnuto. Prije ulaza u algoritam, izvorna poruka dopunjava se nulama tako da njezina duljina u znakovima bude višekratnik broja 48. Nakon toga poruci se dodaje još 40 nula i 64-bitni broj koji predstavlja duljinu poruke u bitovima. Duljina samog broja je 8 okteta, što znači da je poruci ukupno dodan još jedan blok od 48 okteta.

U radu algoritma se koristi 16 S-box tablica veličine 256 riječi (ovdje označena sa Sbox[16][256]) i jedna tablica rotacije (s[i]). Prilikom obrade svakog bloka koristi se i pomoćni spremnik za 16 riječi, a trenutna vrijednost sažetka čuva se u 4 32-bitna spremnika (A, B, C, D).

```
Za i = 0 do (N / 16) – 1:
       X[0..15] = M[i * 16 .. i * 16 + 15].

       Za a = 0 do 7:
         Za b = 0 do 3:
           Za j = 0 do 15:
              t = Sbox[2 * a + (j / 2) mod 2][X[j] mod 256].
              X[(j + 15) mod 16] = X[(j + 15) mod 16] xor t.
              X[(j + 1) mod 16] = X[(j + 1) mod 16] xor t.
           Kraj j.
           Za j = 0 do 15:
              X[j] = rotr(X[j], s[b]).
           Kraj j.
         Kraj b.
       Kraj a.

       A = A xor X[15].
       B = B xor X[14].
       C = C xor X[13].
       D = D xor X[12].
Kraj i.
```

Ispis 8: Funkcija sažimanja algoritma Snefru

Izlaz iz algoritma su vrijednosti A, B, C i D pretvorene u niz okteta prema *big-endian* notaciji.

7.2.8. CRC-32

CRC (*Cyclic Redundancy Code*) algoritmi se često koriste za računanje zaštitne sume, primjerice u telekomunikacijama (otkrivanje greške u prijenosu paketa ili datoteke). CRC algoritmi temelje se na dijeljenju polinoma. Ulazna datoteka duljine *n* bitova promatra se kao binarni polinom stupnja *n*. Računa se ostatak dijeljenja tog polinoma sa polinomom *g(x)* koji je definiran standardom:

CRC-12	$g(x) = x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
CRC-16	$g(x) = x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$g(x) = x^{16} + x^{12} + x^5 + 1$
CRC-32	$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$

Iako vrlo korisni za otkrivanje grešaka, CRC se ne mogu koristiti u kriptografske svrhe, jer nisu jednosmjerne funkcije (zapravo spadaju u klasu linearnih funkcija). Trivijalno je stvoriti drugu poruku koja će imati istu CRC vrijednost kao i izvorna poruka. Ovdje prikazani CRC-32, koji daje 32-bitnu zaštitnu sumu, predstavljen je zbog potpunosti, a ne kao preporuka korištenju u kriptografske svrhe.

Iako je CRC definiran svojim polinomom, algoritam se zbog brzine obično ostvaruje pomoću tablice konstanti (koja je sama generirana iz odgovarajućeg polinoma). Algoritam radi nad pojedinim znakovima poruke.

CRC-32 algoritam koji koristi *lookup* tablicu veličine 256 32-bitnih riječi (CRCTable) glasi:

```
crc = 0xffffffff.

Za i = 0 do N – 1:
      crc = shr(crc, 8) xor CRCTable[(crc mod 256) xor M[i]].
Kraj i.
```

Ispis 9: CRC-32 algoritam

7.2.9. Adler-32

Adler-32 je algoritam za računanje zaštitne sume (*checksum*) podataka koji se koristi u ZLIB sažimanju. Autor mu je Mark Adler, a objavljen je u sklopu specifikacije ZLIB formata podataka u [10].

Kao prednost u odnosu na CRC-32 autor navodi veću brzinu, što je točno ako se CRC ostvaruje dijeljenjem polinoma, no upitno ako CRC-32 algoritam koristi tablicu konstanti (što je gotovo uvijek slučaj). Ovaj algoritam nije pogodan za kriptografsku uporabu iz istih razloga kao i CRC.

Adler-32 koristi dva spremnika trenutnih vrijednosti veličine 16 bita. Krajnji rezultat dobiva se spajanjem tih spremnika. Algoritam u pseudokodu glasi:

```
s1 = 0x0001.
s0 = 0x0000.

Za i = 0 do N – 1:
       s1 = (s1 + M[i]) mod 65521.
       s2 = (s2 + s1) mod 65521.
Kraj i.

rezultat = shr(s2, 16) or s1.
```

Ispis 10: Adler-32 algoritam

7.3. Simetrični algoritmi za kriptiranje

7.3.1. Načini rada simetričnih blok-algoritama

Simetrični blok-algoritmi kriptiraju i dekriptiraju podatke u blokovima od po nekoliko okteta (tipično 8 ili 16 okteta, odnosno 64 ili 128 bita). Kako ti blokovi nisu međusobno ovisni, moguće je modificirati poruku bez poznavanja njenog sadržaja njihovim razmještanjem. Osim toga, ovi algoritmi se ne mogu koristiti kad je potrebno kriptirati oktet po oktet (interaktivne uporabe).

Da bi se riješili ovi problemi, definirano je nekoliko načina rada simetričnih blok-algoritama (prvotno su objavljeni u specifikaciji DES-a):

- *Electronic Cookbook* (ECB)
  Svaki blok se kriptira zasebno.
  Enkripcija: $Ci = E(P_i)$
  Dekripcija: $P_i = D(C_i)$

- *Cipher-block Chaining* (CBC)
  U kriptiranju trenutnog bloka se koristi prethodni. Osigurava međusobnu zavisnost blokova, ali i propagaciju grešaka prilikom enkripcije/dekripcije.
  Enkripcija: $C_i = E(P_i \text{ xor } C_{i-1})$; $C_0 = IV$
  Dekripcija: $P_i = D(C_i) \text{ xor } C_{i-1}$; $C_0 = IV$

- *Cipher feedback* (CFB)
  Koristi se kad je potrebno kriptirati manje od cijelog bloka (npr. samo *r* bitova).
  Enkripcija: $O_i = E(I_i)$; $C_i = P_i \text{ xor left}(O_j, r)$; $I_{i+1} = \text{shl}(I_i, r) + C_i$; $I_1 = IV$
  Dekripcija: $O_i = E(I_i)$; $P_i = C_i \text{ xor left}(O_j, r)$; $I_{i+1} = \text{shl}(I_i, r) + P_i$; $I_1 = IV$

- *Output feedback* (OFB)
  Sličan CFB-u, ali sprečava propagaciju grešaka prilikom enkripcije.
  Enkripcija: $O_i = E(I_i)$; $C_i = P_i \text{ xor left}(O_j, r)$; $I_{i+1} = O_i$; $I_1 = IV$
  Dekripcija: $O_i = E(I_i)$; $P_i = C_i \text{ xor left}(O_j, r)$; $I_{i+1} = O_i$; $I_1 = IV$

7.3.2. DES

DES (Data Encryption Standard) je najpoznatiji algoritam za simetrično kriptiranje podataka, prvi koji je službeno prihvaćen kao standard. Razvijen je sredinom 1970-ih u IBM-u, a nakon toga prihvaćen je kao američki standard (od strane NIST-a te nešto kasnije i ANSI-a) za kriptiranje podataka. Definiran je dokumentom [11]. DES se smatra namjerno oslabljenim (nazivna veličina ključa je 64 bita, ali se koristi samo 56), probijen je i ne preporuča ga se koristiti u kriptografske svrhe. Ovdje je prikazan zbog povijesnih razloga, te zato što mnogi noviji algoritmi koriste zamisli prvi put korištene u njemu.

DES je prvi algoritam koji je koristio ideju *Feistel* mreže, koja je nakon toga korištena i u mnogim drugim simetričnim blok algoritmima. Kod *Feistel* mreže svaki blok podataka podijeli se na dva dijela, "lijevi" i "desni". Enkripcija se provodi u više koraka u kojima se nad jednom polovicom primjenjuje neka funkcija f (koja sama po sebi ne mora biti reverzibilna), nakon čega se blokovi zamjenjuju. Dekripcija se provodi na isti način, samo što se koraci odbrojavaju od zadnjeg do prvog (ključevi se koriste u inverznom poretku). Ideja *Feistel* mreže je povećanim brojem koraka pojačati relativno slabu kriptografsku funkciju f.

DES koristi blokove od 64 bita, podijeljene na dva 32bitna bloka (L i D), uz efektivnu širinu ključa od 56 bita. Broj koraka je 16, a svaka korak koristi 48-bitni ključ (K_i) generiran iz izvornog ključa. U svakom koraku nad 32bitnim podatkom primjenjuje se fiksna funkcija ekspanzije E, 6-na-4-bita supstitucijska funkcija S_i (pažljivo odabrani *S-box*-ovi) i permutacijska funkcija P:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \text{ xor } f(R_{i-1}, K_i)$$
$$f(R_{i-1}, K_i) = P(S_i(E(R_{i-1}) \text{ xor } K_i))$$

Prije prvog koraka, nad blokom podataka obavlja se inicijalna permutacija (*IP*), a nakon posljednje, dijelovi bloka se opet izmjenjuju i obavlja se inverz inicijalne permutacije. Dekripcija se obavlja istim algoritmom, samo što se ključevi i *S-box*-ovi koriste u obrnutom redoslijedu.

Generiranje ključeva za korake K_i obavlja se tako da se iz izvornog ključa na osnovu tablica PC1 i PC2 (*Permuted Choice*) stvaraju 28-bitni registri C i D, koji se u 16 iteracija pomiču za 1 ili 2 bita, ovisno o iteraciji. Na kraju se iz spojenog rezultata uzima 48 bita za ključ K_i.

7.3.3. DESX

DESX je simetrični blok-algoritam koji proširuje DES na duljinu ključa od 120 bita. Dizajniran je od strane RSA Data Security, a službeno nikad nije javno objavljen. Neformalni (ali točan) opis algoritma pojavio se 1994. godine na Usenet grupi sci.crypto.

DESX koristi ključ od 128 bita, od kojih se 64 bita koristi kao ključ za DES (od kojih je zapravo iskorišteno 56 bita – zbog toga je efektivna duljina DESX ključa 120 bita). Drugih 64 bita koristi se u promjeni bloka podataka prije kriptiranja (*pre-Whitening*) i nakon kriptiranja (*post-Whitening*), za što se koristi najobičnija XOR (ekskluzivno-ili) operacija. Na ovaj način bitno se otežava probijanje algoritma.

Prvih 64 bita ključa koristi se izravno za DES, a drugih 64 bita koristi se kao ključ za *pre-Whitening* fazu. Ključ za *post-Whitening* fazu generira se na osnovu ovih ključeva i supstitucijske tablice prema slijedećem algoritmu:

```
DESKey = prva_64_bita(Key)
PreWhitening = druga_64_bita(Key)

PostWhitening = 0
Hash(Input)
Hash(PreWhitening)

funkcija Hash(Ulaz):
       Za svaki okteta Ibyte od Ulaz:
              indeks = prvih_8_bita(PostWhitening) xor
                     drugih_8_bita(PostWhitening)
              zadnji = DESXTablica[indeks]
              shl(PostWhitening, 8)
              zadnjih_8_bita(PostWhitening) = zadnji xor Ibyte
       Kraj Ibyte.
Kraj.
```

Ispis 11: DESX algoritam

Prilikom kriptiranja, 64-bitni blok se prvo XOR-a PreWhitening ključem, a nakon enkripcije PostWhitening ključem. Kod dekripcije prvo se koristi PostWhitening ključ a nakon toga PreWhitening ključ.

7.3.4. TripleDES

TripleDES (poznat i kao 3-DES i TDEA) je simetrični blok-algoritam za enkripciju koji također predstavlja proširenje DES algoritma. Nastao je kad je postalo očito da DES ne pruža dobru zaštitu, a trenutačno nije bilo dostupnih boljih enkripcijskih algoritama. Sam TripleDES se smatra potpuno sigurnim od *brute-force* napada, ali je vrlo spor (tri puta je sporiji od DES-a). Algoritam su predložili W. Diffie, M, Hellman i W. Tuchmann, a prihvaćen je kao standard i definiran dokumentom [11].

TripleDES koristi 192-bitni ključ (koji se dijeli u 3 64-bitna dijela), a blok podataka kriptira i dekriptira koristeći standardni DES algoritam, tri puta zaredom. Ako se jasni tekst označi sa *P*, kriptirani tekst sa *C*, a ključ sa $K_1K_2K_3$, algoritam glasi:

C = DES_Enkripcija(DES_Dekripcija(DES_Enkripcija(P, K_1), K_2), K_3)
P = DES_Dekripcija(DES_Enkripcija(DES_Dekripcija(C, K_3), K_2), K_1)

Kako jedna TripleDES operacija više puta koristi DES algoritam, posebno su definirani načini rada TripleDES algoritma (*TDEA modes of operation*). TECB, TCBC, TCFB i TOFB načini rada samo se preslikavaju u ECB, CBC, CFB i OFB načine rada unutarnjeg DES algoritma.

7.3.5. AES

AES (Advanced Encryption Standard) je simetrični blok-algoritam odabran od NIST-a kao novi standard ([12]) za kriptiranje podataka. AES, izvorno nazvan Rijndael, odabran je kao pobjednik natječaja za izradu novog algoritma koji neće imati mane DES-a. Postoji više inačica Rijndaela, a kao AES se uzima inačica koja koristi 128-bitni ključ i radi nad 128-bitnim blokovima podataka. Broj koraka ovisi o veličini bloka podataka i veličini ključa – za AES, koristi se 10+1 koraka.

Za razliku od DES-a i njegovih derivata, AES nije temeljen na Feistel strukturi. Njegovi koraci koriste 4 uniformno reverzibilne transformacije (blok podataka organiziran je u tablicu 4 puta 4 znaka):

- *SubBytes* – nelinearna supstitucija oktet po oktet pomoću S-boxova,
- *ShiftRows* – ciklički pomak okteta u zadnja 3 reda bloka,
- *MixColumn* – svaki stupac bloka se tretira kao polinom nad GF(2^8) i množi polinomom modulo $x^4 + 1$,
- *AddRoundKey* – ključ koraka dodaje se bloku XOR operacijom.

```
AES_Kriptiranje(oktet ulaz[16], oktet izlaz[16], rijec w[44]):
       oktet stanje[4,4].
       stanje = ulaz.

       AddRoundKey(stanje, w[0,3]).
       Za i = 1 do 9:
              SubBytes(stanje).
              ShiftRows(stanje).
              MixColumns(stanje).
              AddRoundKey(stanje, w[i * 4, i * 4 + 3]).
       Kraj i.
       SubBytes(stanje).
       ShiftRows(stanje).
       AddRoundKey(stanje, w[40, 43]).
       izlaz = stanje.
Kraj.

AES_GeneriranjeKljuca(oktet kljuc[16], rijec w[44]):
       rijec temp.

       w[0] = kljuc[0,3].
       w[1] = kljuc[4,7].
       w[2] = kljuc[8,11].
       w[3] = kljuc[12,15].
       i = 4.
       Dok i < 44:
              temp = w[i - 1].
              Ako i % 4 = 0:
                     temp = SubWord(RotWord(temp)) xor Rcon[i / 4].
              Kraj.
              w[i] = w[i - 4] xor temp.
              i = i + 1.
       Kraj i.
Kraj.
```

Ispis 12: AES algoritam

Dekripcija se provodi analogno enkripciji, samo što se koraci provode u obrnutom redoslijedu, i koriste se inverzni ključevi.

7.3.6. IDEA

IDEA (International Data Encryption Algorithm) razvijen je 1992., a njegovi tvorci su Xueija Lai i James Massey (referenca: [13]). Algoritam je temeljen na generalizaciji *Feistel* strukture, a sastoji se od 8 identičnih koraka i izlazne transformacije.

Dominantni koncept u dizajnu algoritma je mješanje operacija različitih algebarskih grupa (blokovi *a*, *b*, *c* su 16-bitni):

- *XOR po bitovima*: c = a XOR b
- *zbrajanje modulo* 2^n: c = (a + b) % 65536 (ovdje označeno sa: ADDX)
- *modificirano množenje modulo* 2^n+1 *uz* $0 \in \boldsymbol{Z}_{2^n} = 2^n \in \boldsymbol{Z}_{2^{n+1}}$ (ovdje označeno sa: MULX)

Algoritam radi nad 64-bitnim blokovima podataka; ulazni blok dijeli se na 4 16-bitna dijela nad kojima se obavljaju sve operacije. Prilikom rada koristi se u svakom koraku 6 različitih podključeva, te još 4 podključa u izlaznoj transformaciji.

```
IDEA_Kriptiranje(oktet ulaz[8], oktet izlaz[8], rijec kljuc[6 * 8 + 4]):
       rijec x[4] = ( ulaz[0,1], ulaz[2,3], ulaz[3,4], ulaz[4,5] ).

       Za r = 0 do 7:
              x[0] = x[0] MULX k[6 * r].
              x[3] = x[3] MULX k[6 * r + 3].
              x[1] = x[1] ADDX k[6 * r + 1].
              x[2] = x[2] ADDX k[6 * r + 2].
              t0 = k[6 * r + 4] MULX (x[0] XOR x[2]).
              t1 = k[6 * r + 5] MULX (t0 ADDX (x[1] XOR x[3])).
              t2 = t0 ADDX t1.
              x[0] = x[0] XOR t1.
              x[3] = x[3] XOR t2.
              a = x[1] XOR t2.
              x[1] = x[2] XOR t1.
              x[2] = a.
       Kraj r.

       izlaz[0,1] = x[0] MULX k[6 * 8].
       izlaz[6,7] = x[3] MULX k[6 * 8 + 3].
       izlaz[2,3] = x[2] ADDX k[6 * 8 + 1].
       izlaz[4,5] = x[1] ADDX k[6 * 8 + 2].
Kraj.

IDEA_GeneriranjeKljuca(oktet kljuc[16], rijec k[6 * 8 + 4]):

       k[0] = kljuc[0,1] ... k[7] = kljuc[14,15].

       Dok nisu_generirani_svi_podkljucevi:
              kljuc = rotl(kljuc, 25).
              k[i] = kljuc[0,1] ... k[i + 7] = kljuc[14,15]
       Kraj.
Kraj.
```

Ispis 13: IDEA algoritam

Dekripcija se provodi po istom algoritmu kao i enkripcija, ali se kao podključevi koriste inverzi ključeva korištenih pri enkripciji. Za njihovo generiranje prvo je potrebno generirati enkripcijske ključeve, koji se nakon toga na osnovu nekoliko formula transformiraju u dekripcijske ključeve.

7.3.7. Blowfish

Blowfish algoritam razvio je 1993. godine Bruce Schneier (specificiran je u [14]), a danas je jedna od najpopularnijih alternativa DES algoritmu. Osnovna svojstva algoritma su jednostavno ostvarenje, velika brzina izvođenja i mali memorijski zahtjevi.

Blowfish algoritam je *Feistel* mreža koja se sastoji od 16 koraka, a radi nad 64-bitnim blokovima podataka. Duljina ključa je varijabilna (minimalno 32, maksimalno 448 bitova), a u procesu enkripcije i dekripcije koristi se 16 32-bitnih podključeva i 4 32-bitna S-boxa od po 256 zapisa, koji se svi generiraju iz danog ključa. Podključevi i S-boxovi generiraju se samim Blowfish algoritmom, za što je potrebno 521 iteracija.

Algoritam je optimiran za izvođenje na 32-bitnim procesorima, pa su sve korištene operacije modulo 2^{32}. U pretvaranju između 32-bitnih registara i okteta koristi se *big-endian* notacija. Dekripcija je identična enkripciji, uz obrnut redoslijed podključeva.

```
F(rijec x):
	oktet (a, b, c, d) = x.
	izlaz = ((s[0][a] + s[1][b]) xor s[2][c]) + s[3][d].
Kraj.

Blowfish_Kriptiranje(oktet ulaz[8], oktet izlaz[8], rijec p[18], rijec s[4][256]):
	rijec xl = ulaz[0..3].
	rijec xr = ulaz[4..7].
	Za r = 0 do 15:
		xl = xl xor p[r].
		xr = F(xl) xor xr.
		zamijeni(xl, xr).
	Kraj r.

	zamijeni(xl, xr).
	xr = xr xor p[16].
	xl = xl xor p[17].
	izlaz[0..3] = xl.
	izlaz[4..7] = xr.
Kraj.

// P i S su inicijalizirani unaprijed definiranim nizom okteta
Blowfish_GeneriranjeKljuca(oktet kljuc[], rijec p[18], rijec s[4][256]):
	oktet prosireni_kljuc = kljuc + kljuc + ...
	oktet blok[8] = ( 0, 0, 0, 0, 0, 0, 0, 0 ).

	Za i = 0 do 17:
		p[i] = p[i] xor prosireni_kljuc[i * 4 .. i * 4 + 3].
	Kraj i.
	Za i = 0 do 9:
		blok = Blowfish_Kriptiranje(blok, blok, p, s).
		p[2 * i] = blok[0 .. 3].
		p[2 * i + 1] = blok[4 .. 7].
	Kraj i.
	Za i = 0 do 3:
		Za j = 0 do 127:
			blok = Blowfish_Kriptiranje(blok, blok, p, s).
			s[i][2 * j] = blok[0 .. 3].
			s[i][2 * j + 1] = blok[4 .. 7].
		Kraj j.
	Kraj i.
Kraj.
```

Ispis 14: Blowfish algoritam

7.3.8. DEAL

DEAL (Data Encryption Algorithm with Larger blocks) je 128-bitni simetrični blok enkripcijski algoritam temeljen na DES-u sa ključevima duljine 128, 192 ili 256 bitova. Osmislio ga je 1998. Lars R. Knudsen ([15]) kao zamjenu za DES koja je boljih performansi nego TripleDES.

Sam algoritam je *Feistel* mreža sa 6 ili 8 koraka koja kao enkripcijsku funkciju koristi DES enkripciju, koristeći u svakom koraku drugačiji ključ enkripcije. Ulazni 128-bitni blok podataka dijeli se na dva 64-bitna dijela, X^L i X^R, a postupak u svakom koraku *j* je:

$X^L_j = E_{RKj}(X^L_{j-1})$ XOR X^R_{j-1}
$X^R_j = X^L_{j-1}$

U inačici algoritma koji koristi 128-bitne ključeve i 6 koraka, izvorni ključ rastavlja se na dva 64-bitna ključa K_1 i K_2, a podključevi $RK_1..RK_6$ dobivaju se prema formulama:

$K = (0123456789abcdef)_{16}$
$RK_1 = E_K(K_1)$
$RK_2 = E_K(K_2 \text{ xor } RK_1)$
$RK_3 = E_K(K_1 \text{ xor } 1 \text{ xor } RK_2)$
$RK_4 = E_K(K_2 \text{ xor } 2 \text{ xor } RK_3)$
$RK_5 = E_K(K_1 \text{ xor } 4 \text{ xor } RK_4)$
$RK_6 = E_K(K_2 \text{ xor } 8 \text{ xor } RK_5)$

7.3.9. RC2

RC2 ("*Rivest Code 2*") enkripcijski algoritam razvio je Ron Rivest, a specificiran je u [16]. Algoritam radi sa 64-bitnim blokovima podataka i koristi 64-bitne ključeve, a optimiran je za 16-bitne procesore. Ovaj algoritam je zastario i probijen je pa ga se ne preporuča koristiti u kriptografske svrhe, a ovdje je prikazan iz povijesnih razloga.

Algoritam koristi 64 podključa veličine 16 bita, a u njihovoj generaciji kao i samoj enkripciji i dekripciji koristi se i tablica konstanti veličine 256 okteta. Pretvaranje okteta u 16-bitne riječi i obrnuto obavlja se po *little-endian* notaciji.

```
RC2_GeneriranjeKljuca(oktet kljuc[8], rijec k[64]):
	oktet L[128].

	L[0..7] = kljuc[0..7].
	Za i = 8 do 127:
		L[i] = pitable[(L[i - 1] + L[i - 8]) mod 256].
	Kraj i.
	L[120] = pitable[L[120] mod 256].
	Za i = 119 do 0:
		L[i] = pitable[L[i + 1] xor L[i + 8]].
	Kraj i.
	Za i = 0 do 63:
		k[i] = L[i * 2, i * 2 + 1].
	Kraj i.
Kraj.

mixup(i):
	R[i] = R[i] + K[j] + (R[i-1] and R[i-2]) + ((not R[i-1]) and R[i-3]).
	j = j + 1.
	R[i] = rotl(R[i], s[i]).
Kraj.

mixround():
	mixup(0). mixup(1). mixup(2). mixup(3).
Kraj.

mash(i):
	R[i] = R[i] + K[R[i-1] and 63].
Kraj.

mashround():
	mash(0). mash(1). mash(2). mash(3).
Kraj.

RC2_Kriptiranje(oktet ulaz[16], rijec k[64]):
	j = 0.
	Za i = 0 do 4: mixround().
	mashround().
	Za i = 0 do 5: mixround().
	mashround().
	Za i = 0 do 4: mixround().
Kraj.
```

Ispis 15: RC2 algoritam

Dekripcija se obavlja analogno enkripciji, koristeći iste pomoćne ključeve, ali u obrnutom redoslijedu, i inverze operacija *mixup* i *mash*, također u obrnutom redoslijedu.

7.3.10. RC4

RC4 je simetrični tok-orjentirani kriptografski algoritam, što znači da ne djeluje nad blokovima podataka kao ostali ovdje opisani algoritmi, nego nad pojedinačnim oktetima. Autor mu je RSA Data Security, a sam algoritam je zaštićen i tajan. No, na Internet je 1994. godine "procurio" opis algoritma ArcFour ([17]) uskladivog (u smislu davanja istog izlaza za isti ulaz i ključeve) ovom algoritmu.

RC4 se temelji na posmačnom registru sa linearnom povratnom vezom (LFSR – *Linear Feedback Shift Register*), odnosno generira niz pseudoslučajnih brojeva koji se XOR operacijom kombiniraju sa svakim znakom ulaza kako bi se dobio izlaz. Dekripcija se provodi na identičan način, pošto je XOR operacija sa istim argumentom sama sebi inverz.

```
RC4_Inicijalizacija(oktet kljuc[]):
       oktet s[256].

       oktet k[256] = kljuc + kljuc + ...
       Za i = 0 do 255: s[i] = i.

       j = 0.
       Za i = 0 do 255:
              j = (j + s[i] + k[i]) mod 256;
              zamijeni(s[j], s[i]).
       Kraj i.
Kraj.

RC4_Kriptiranje(oktet ulaz[], oktet izlaz[], duljina):
       i = 0.
       j = 0.

       Za k = 0 do duljina:
              i = (i + 1) mod 256.
              j = (j + s[i]) mod 256.
              zamijeni(s[i], s[j]).
              t = s[(s[i] + s[j]) mod 256].

              izlaz[k] = ulaz[k] xor t.
       Kraj k.
Kraj.
```

Ispis 16: RC4 algoritam

7.3.11. RC5

RC5 enkripcijski algoritam je simetrični blok enkripcijski algoritam varijabilne duljine ključa, bloka podataka i broja koraka. Autor mu je Ron Rivest, a objavljen je 1994. godine u publikaciji [18]. Koncepcija algoritma zasniva se na rotacijama ovisnima o ulaznim rotacijama. Kao i RC2, i ovaj algoritam je probijen pa ga se ne preporuča koristiti u kriptografske svrhe, a ovdje je prikazan iz povijesnih razloga.

Algoritam je vrlo kompaktan i jednostavan za ostvarenje u softveru ili sklopovlju, a najčešće korišteni parametri su 128-bitni ključ i 12 koraka, što je pogodno za ostvarenje na 32-bitnim procesorima (oznaka algoritma: RC5-32/12/16). Duljina bloka podataka je 64 bita, a pretvaranje iz okteta u riječi i obrnuto obavlja se po *little-endian* notaciji.

```
RC5_Kriptiranje(rijec blok[2], rijec kljuc[22]):
	blok[0] = blok[0] + kljuc[0].
	blok[1] = blok[1] + kljuc[1].

	Za i = 1 do 12:
		blok[0] = rotl(blok[0] xor blok[1], blok[1]) + kljuc[2 * i].
		blok[1] = rotr(blok[1] xor blok[0], blok[0]) + kljuc[2 * i + 1].
	Kraj i.
Kraj.

RC5_Dekriptiranje(rijec blok[2], rijec kljuc[22]):
	Za i = 12 do 1:
		blok[1] = rotr(blok[1] – kljuc[2 * i + 1], blok[0]) xor blok[0].
		blok[0] = rotr(blok[0] – kljuc[2 * i], blok[1]) xor blok[1].
	Kraj i.

	blok[0] = blok[0] - kljuc[0].
	blok[1] = blok[1] - kljuc[1].
Kraj.

RC5_GeneriranjeKljuca(oktet kljuc[16], oktet k[12]):
	rijec L[4].

	L[0] = kljuc[0..3] .. L[3] = kljuc[14..15].
	S[0] = 0xb7e15163.
	Za i = 1 do 25: S[i] = S[i – 1] + 0x9e3779b9.

	i = 0. j = 0. A = 0. B = 0.
	Za k = 0 do (3 * 26 – 1):
		A = rotl(k[i] + A + B, 3).
		k[i] = A.
		B = rotl(L[j] + A + B, A + B).
		L[j] = B.
		i = (i + 1) mod 26.
		j = (j + 1) mod 4.
	Kraj k.
Kraj.
```

Ispis 17: RC5 algoritam

7.3.12. RC6

RC6 algoritam nastao je 1998. godine razvojem algoritma RC5 u skladu s zahtjevima NIST-ovog natječaja za AES. Autor mu je Ron Rivest, a specificiran je u [19].

RC6 također koristi ideju rotacija ovisnih o ulaznim podacima, a novosti su korištenje 4 registra umjesto 2 i korištenje cjelobrojnog množenja. Algoritam radi sa 128-bitnim ključevima i 128-bitnim blokovima podataka (verzija RC6-32/20/16, uskladiva s AES specifikacijom), no sve operacije obavljaju se nad 32-bitnim riječima, što ovaj algoritam čini pogodnim za ostvarenje na 32-bitnim arhitekturama.

```
RC6_Kriptiranje(oktet ulaz[16], oktet izlaz[16], rijec kljuc[44]):
	rijec a, b, c, d.
	a = ulaz[0..3]. b = ulaz[4..7]. c = ulaz[8..11]. d = ulaz[12..15].
	b = b + kljuc[0]. d = d + kljuc[1].
	Za i = 1 do 20:
		t = rotl(b * (2 * b + 1), 5).
		u = rotl(d * (2 * d + 1), 5).
		a = rotl(a xor t, u) + kljuc[2 * i].
		c = rotl(c xor u, t) + kljuc[2 * i + 1].
		t = a. a = b. b = c. c = d. d = t.
	Kraj i.
	a = a + kljuc[42].  c = c + kljuc[43].
	izlaz[0..3] = a. izlaz[4..7] = b. izlaz[8..11] = c. izlaz[12..15] = d.
Kraj.

RC6_Deriptiranje(oktet ulaz[16], oktet izlaz[16], rijec kljuc[44]):
	rijec a, b, c, d.
	a = ulaz[0..3]. b = ulaz[4..7]. c = ulaz[8..11]. d = ulaz[12..15].
	c = c - kljuc[43]. a = a - kljuc[42].
	Za i = 20 do 1:
		t = d. d = c. c = b. b = a. a = t.
		u = rotl(d * (2 * d + 1), 5).
		t = rotl(b * (2 * b + 1), 5).
		c = rotr(c - kljuc[2 * i + 1], t) xor u.
		a = rotr(a - kljuc[2 * i], u) xor t.
	Kraj i.
	d = d + kljuc[1]. a = a + kljuc[0].
	izlaz[0..3] = a. izlaz[4..7] = b. izlaz[8..11] = c. izlaz[12..15] = d.
Kraj.

RC6_GeneriranjeKljuca(oktet kljuc[16], oktet k[22]):
	rijec L[4].
	L[0] = kljuc[0..3] .. L[3] = kljuc[14..15].
	S[0] = 0xb7e15163.
	Za i = 1 do 43: S[i] = S[i - 1] + 0x9e3779b9.
	i = 0. j = 0. A = 0. B = 0.
	Za k = 0 do (3 * 44 - 1):
		A = rotl(k[i] + A + B, 3).
		k[i] = A.
		B = rotl(L[j] + A + B, A + B).
		L[j] = B.
		i = (i + 1) mod 44.
		j = (j + 1) mod 4.
	Kraj k.
Kraj.
```

Ispis 18: RC6 algoritam

7.3.13. ThreeWay

ThreeWay (3-Way) je simetrični blok kriptografski algoritam nastao 1993. godine ([20]). Autor mu je Joan Daemen (poznatiji kao jedan od autora AES-a). Algoritam radi nad 96-bitnim blokovima podataka i koristi 96-bitne ključeve, a same operacije provode se nad 32-bitnim riječima.

ThreeWay definira nekoliko transformacija koje se paralelno obavljaju nad svim riječima kako prilikom pripreme ključeva tako i prilikom same enkripcije i dekripcije. To su:

- μ – inverzija bitova u riječi
- γ – nelinearna transformacija
- θ – linearna transformacija
- π – permutacija
- ρ – korak algoritma: $\rho(x) = \pi_2(\gamma(\pi_1(\theta(x))))$

```
ThreeWay_Inicijalizacija(oktet kljuc[12]):
       k[0] = inv_k[0] = kljuc[0..3].
       k[1] = inv_k[1] = kljuc[4..7].
       k[2] = inv_k[2] = kljuc[8..11].

       theta(inv_k).
       mu(inv_k).
Kraj.

// rijec rcon[12] – permutacijske konstante za enkripciju
ThreeWay_Kriptiranje(rijec blok[3], rijec k[3]):
       Za i = 0 do 10:
              blok[0] = blok[0] xor k[0] xor (rcon[i] * 65536).
              blok[1] = blok[1] xor k[1].
              blok[2] = blok[2] xor k[2] xor rcon[i].
              rho(blok).
       Kraj i.
       blok[0] = blok[0] xor k[0] xor (rcon[11] * 65536).
       blok[1] = blok[1] xor k[1].
       blok[2] = blok[2] xor k[2] xor rcon[11].
       theta(blok).
Kraj.

// rijec rcon[12] – permutacijske konstante za dekripciju
ThreeWay_Deriptiranje(rijec blok[3], rijec k[3]):
       mu(blok).
       Za i = 0 do 10:
              blok[0] = blok[0] xor k[0] xor (rcon[i] * 65536).
              blok[1] = blok[1] xor k[1].
              blok[2] = blok[2] xor k[2] xor rcon[i].
              rho(blok).
       Kraj i.
       blok[0] = blok[0] xor k[0] xor (rcon[11] * 65536).
       blok[1] = blok[1] xor k[1].
       blok[2] = blok[2] xor k[2] xor rcon[11].
       theta(blok).
       mu(blok).
Kraj.
```

Ispis 19: ThreeWay algoritam

7.4. Asimetrični kriptografski algoritmi

7.4.1. RSA

RSA kriptografski sustav (nazvan po svojim tvorcima: R. Rivest, A. Shamir, L. Adleman) je najpoznatiji i najkorišteniji asimetrični kriptografski sustav. Objavljen 1978., ovaj algoritam dugo je bio patentiran, no nedavno je patent istekao pa se RSA može smatrati *de facto* standardom na ovom području. Moguće ga je koristiti u enkripciji, digitalnim potpisima i razmjeni ključevima, a sigurnost mu se temelji na problemu faktorizacije vrlo velikih brojeva.

Algoritam generiranja ključeva u RSA sustavu je:

1. generirati dva velika različita prosta broja p i q otprilike jednake veličine,
2. izračunati $n = pq$ i $z = (p - 1)(q - 1)$,
3. odabrati slučajni cijeli broj e ($1 < e < z$) tako da e i z budu relativno prosti,
4. izračunati cijeli broj d ($1 < d < z$) takav da $ed \equiv 1 \pmod{z}$,
5. par brojeva (n, e) je javni ključ; par brojeva (n, d) je privatni ključ.

Algoritam enkripcije glasi:

1. predstaviti poruku kao cijeli broj m iz intervala $[0, n - 1]$,
2. izračunati $c = m^e \bmod n$,
3. c je kriptirana poruka.

Dekripcija je analogna i glasi:

4. predstaviti kriptirani tekst kao cijeli broj c iz intervala $[0, n - 1]$,
5. izračunati $m = c^d \bmod n$,
6. m je izvorna poruka (jasni tekst).

Kako je sam algoritam simetričan (ovdje pojam “simetričan” znači samo da se koriste iste operacije i za enkripciju i za dekripciju), moguće je jasni tekst kriptirati privatnim ključem a dekriptirati javnim, što se upotrebljava u digitalnom potpisivanju.

7.4.2. ElGamal

ElGamal asimetrični kriptografski sustav temelji se na problemu pronalaženja diskretnog logaritma, a jedno od svojstava mu je i korištenje randomizacije kako bi se povećala razina sigurnosti. Autor mu je T. ElGamal, a sustav svoju popularnost može dijelom zahvaliti i činjenici da je RSA dugo vrijeme bio patentiran algoritam. Nedostatak ovog sustava je ekspanzija poruke, odnosno kriptirani tekst je dvostruko dulji od jasnog teksta.

Algoritam generiranja ključeva u ElGamal sustavu je:

1. generirati veliki slučajni prosti broj p i generator α multiplikativne grupe $\boldsymbol{Z}_p^*$ cijelih brojeva modulo p,
2. odabrati slučajni cijeli broj a, $1 \leq a \leq p-2$ i izračunati $\alpha^a \bmod p$,
3. javni ključ je (p, α, α^a); privatni ključ je a.

Postupak enkripcija poruke je:

1. predstaviti poruku kao cijeli broj m iz intervala $[0, p-1]$,
2. odabrati slučajni cijeli broj k, $1 \leq k \leq p-2$,
3. izračunati $\gamma = \alpha^a \bmod p$ i $\delta = m \cdot (\alpha^a)^k \bmod p$,
4. kriptirani tekst je (γ, δ).

Algoritam dekripcije je:

1. izračunati γ^{p-1-a} pomoću a ($\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$),
2. izračunati $m = (\gamma^{-a}) \cdot \delta \bmod p$,
3. jasni tekst je m.

7.4.3. LUC

LUC je asimetrični algoritam kriptiranja objavljen 1993. godine u [21]. Analogan je RSA algoritmu i temelji se na modularnoj aritmetici, ali za razliku od RSA koji koristi modularno potenciranje LUC koristi Lucasove funkcije.

Lucasove funkcije spadaju u linearne rekurzivne relacije višeg reda, a njihova svojstva analogna su svojstvima potenciranja pa se mogu koristiti za enkripciju, elektroničnim potpisima i razmjeni ključeva.

Generiranje ključeva u LUC sustavu obavlja se u slijedećim koracima:

1. generirati dva velika različita prosta broja p i q otprilike jednake veličine,
2. izračunati $n = pq$ i $z = (p - 1)(q - 1)(p + 1)(q + 1)$,
3. odabrati slučajni cijeli broj e ($1 < e < z$) tako da e i z budu relativno prosti,
4. izračunati cijeli broj d ($1 < d < z$) takav da $ed \equiv 1 \pmod{z}$,
5. par brojeva (n, e) je javni ključ; par brojeva (n, d) je privatni ključ.

Algoritam LUC enkripcije glasi:

1. predstaviti poruku kao cijeli broj m iz intervala $[0, n - 1]$,
2. izračunati $c = V_e(m,1) \bmod n$ (V_e je Lucasova funkcija),
3. c je kriptirana poruka.

Dekripcija je analogna i glasi:

1. predstaviti kriptirani tekst kao cijeli broj c iz intervala $[0, n - 1]$,
2. izračunati $m = V_d(c,1) \bmod n$,
3. m je izvorna poruka (jasni tekst).

7.4.4. Diffie-Hellman razmjena ključeva

Diffie-Hellman metoda razmjene ključeva (poznata i pod nazivom *exponential key exchange*), objavljena 1976. godine, je prvo praktično rješenje problema raspodjele ključeva, te omogućuje dvjema stranama koje po prvi puta komuniciraju da dogovore zajednički tajni ključ razmjenjujući poruke preko otvorenog i nesigurnog kanala.

Metoda se zasniva na problemu pronalaženja diskretnog logaritma i Diffie-Hellman problemu. Algoritam razmjene ključeva glasi:

1. generirati veliki slučajni prosti broj p i generator α multiplikativne grupe $\boldsymbol{Z}_p^*$ cijelih brojeva modulo p,
2. parametri p i α mogu biti javni i standardizirani,
3. strana A odabere slučajni broj a, $1 \leq a \leq p-2$, izračuna $x = \alpha^a \bmod p$ i pošalje x strani B,
4. strana B odabere slučajni broj b, $1 \leq b \leq p-2$, izračuna $y = \alpha^b \bmod p$ i pošalje y strani A,
5. A prima poruku i računa zajednički ključ $K = y^a \bmod p = \alpha^{ba} \bmod p$,
6. B prima poruku i računa zajednički ključ $K = x^b \bmod p = \alpha^{ab} \bmod p$.

8. Zaključak

U današnje vrijeme kriptografija se sve više koristi, kako za zaštitu i očuvanje integriteta korisničkih podataka ili komunikacije, tako i u samom radu računalnih sustava, primjerice za provjeru programa skinutih s Interneta i zaštitu protiv računalnih virusa.

Prosječni korisnik u svakodnevnom radu koristi različite aplikacije koje koriste kriptografiju (*web browser*, *mail* klijent, instalacijski programi, *secure shell*, ...). Biblioteke kriptografskih funkcija (odnosno kriptografski moduli) omogućuju autorima tih aplikacija jednostavno uključivanje kriptografije i pružanje veće sigurnosti svojim korisnicima.

Prilikom izrade biblioteke koja pruža ostvarenje kriptografskih algoritama potrebno je obratiti pažnju na nekoliko bitnih problema:

- funkcionalnost – biblioteka mora biti potuna u smislu ostvarenja svih algoritama koji su korisniku (programeru) potrebni u njegovoj aplikaciji,
- učinkovitost – kriptografski algoritmi, posebice asimetrični kriptografski sustavi, su vrlo zahtjevni za računanje, pa je prilikom njihovog ostvarivanja potrebno maksimizirati njihovu učinkovitost,
- prenosivost – prenosivost na različite računalne platforme omogućuje korisniku izbor platforme na osnovu njegovih potreba, bez bojazni da će morati tražiti drugu biblioteku ili samostalno ostvarivati potrebne algoritme,
- jednostavnost programskog sučelja – biblioteka služi tome da korisnika (programera) oslobodi bavljenja detaljima algoritama; ako je programsko sučelje nejasno i pretjerano složeno, smanjuje se produktivnost programera,
- otvorenost – otvorenost biblioteke pruža korisniku mogućnost dograđivanja ili izmjene biblioteke, prenošenja na nove računalne platforme, kao i veću vjerojatnost pronalaženja eventualnih grešaka i nedostataka u biblioteci, čime se osigurava i veća pouzdanost u njenom korištenju.

Dodatak A – Cyfer sučelje za programiranje aplikacija (Cyfer C API)

A.1 – Funkcije za izračunavanje sažetka poruke

Potrebna header datoteka: `hash.h`

Get_Supported()

Prototip: `CYFER_Hash_t *CYFER_Hash_Get_Supported();`
Vraća pokazivač na niz struktura koje opisuju podržane algoritme.

Select()

Prototip: `CYFER_Hash_Select(char *name, size_t *length);`
Odabire algoritam pod nazivom 'name' – postavlja duljinu sažetka (u oktetima), i vraća broj koji predstavlja odabrani algoritam. U slučaju greške, funkcija vraća vrijednost CYFER_HASH_NONE.

Init()

Prototip: `CYFER_HASH_CTX *CYFER_Hash_Init(int type);`
Stvara i inicijalizira *hash* kontekst. U slučaju greške (ako 'type' ne predstavlja podržani algoritam, ili neka druga greška) funkcija vraća NULL, inače vraća novi kontekst.

Update()

Prototip: `CYFER_Hash_Update(CYFER_HASH_CTX *ctx, char *buffer, size_t size);`
Obrađuje blok podataka specificirane duljine funkcijom za izračunavanje sažetka, mijenjajući trenutnu vrijednost sažetka poruke. Zbog načina rada algoritma (podjela na blokove), moguće je da cijeli predani blok podatka nije odmah obrađen, nego kopiran na privremeno spremište ("buffer").

Finish()

Prototip: `CYFER_Hash_Finish(CYFER_HASH_CTX *ctx, char *result);`
Završava algoritam (obrađuje preostale podatke ako su ostali u spremniku), sprema sažetak poruke u specificirani spremnik i uništava kontekst. Pretpostavlja se da spremnik ima dovoljno mjesta za onoliko okteta koliko je vratila funkcija Select().

Hash()

Prototip: `CYFER_Hash(CYFER_HASH_CTX *ctx, char *type, char *input, size_t size, char *result);`
Pomoćna funkcija koja izračunava sažetak poruke (algoritmom specificiranom imenom 'type') spremljene u ulaznoj varijabli (čija je duljina navedena u parametru 'size'). Sažetak se sprema u izlazni spremnik za koji se pretpostavlja da ima dovoljno mjesta da primi rezultat. U slučaju greške (nepoznat algoritam), funkcija vraća vrijednost -1, inače vraća vrijednost 0.

A.2 – Funkcije za rad sa simetričnim blok-algoritmima (block ciphers)

Potrebna header datoteka: `cipher.h`

Get_Supported()

Prototip: `CYFER_BlockCipher_t *CYFER_BlockCipher_Get_Supported();`
Vraća pokazivač na niz struktura koje opisuju podržane algoritme.

Get_SupportedModes()

Prototip: `CYFER_BlockMode_t *CYFER_BlockCipher_Get_SupportedModes();`
Vraća pokazivač na niz struktura koje opisuju podržane načine rada simetričnih blok-algoritama.

Select()

Prototip: `int CYFER_BlockCipher_Select(char *name, size_t *keylen, size_t *length);`
Odabire algoritam pod nazivom 'name' – postavlja korištenu duljinu ključa i duljinu bloka podataka. Za algoritme koji koriste promjenjivu duljinu ključa, vrijednost predstavlja maksimalnu duljinu ključa. U slučaju greške, funkcija vraća vrijednost -1.

SelectMode()

Prototip: `int CYFER_BlockCipher_SelectMode(char *name, size_t *length);`
Odabire način rada pod nazivom 'name' – postavlja korištenu duljinu bloka podataka. Ako način rada ne mijenja duljinu bloka korištenog algoritma, vraćena vrijednost duljine je 0. U slučaju greške, funkcija vraća vrijednost -1.

Init()

Prototip: `CYFER_BLOCK_CIPHER_CTX *CYFER_BlockCipher_Init(int type, char *key, size_t size, int mode, char *ivec);`
Stvara kontekst i inicijalizira algoritam sa priloženim ključem, koristeći navedeni način rada. Kod algoritama koji imaju određenu veličinu ključa, veličina ključa se ignorira. Za inicijalizacijski vektor može se koristiti vrijednost NULL, što odabire "prazan" (popunje nulama) blok podataka. U slučaju greške funkcija vraća NULL, inače vraća novi kontekst.

Encrypt()

Prototip: `CYFER_BlockCipher_Encrypt(CYFER_BLOCK_CIPHER_CTX *ctx, char *input, char *output);`
Kriptira blok podataka.

Decrypt()

Prototip: `CYFER_BlockCipher_Decrypt(CYFER_BLOCK_CIPHER_CTX *ctx, char *input, char *output);`
Dekriptira blok podataka.

Finish()

Prototip: `CYFER_BlockCipher_Finish(CYFER_BLOCK_CIPHER_CTX *ctx);`
Završava rad algoritama i uništava kontekst.

A.3 – Funkcije za rad sa simetričnim tok-algoritmima (stream ciphers)

Potrebna header datoteka: `cipher.h`

Get_Supported()

Prototip: `CYFER_StreamCipher_t *CYFER_StreamCipher_Get_Supported();`
Vraća pokazivač na niz struktura koje opisuju podržane algoritme.

Select()

Prototip: `int CYFER_StreamCipher_Select(char *name, size_t *keylen);`
Odabire algoritam pod nazivom 'name' – postavlja korištenu duljinu ključa. Za algoritme koji koriste promjenjivu duljinu ključa, vrijednost predstavlja maksimalnu duljinu ključa. U slučaju greške, funkcija vraća vrijednost -1.

Init()

Prototip: `CYFER_STREAM_CIPHER_CTX *CYFER_StreamCipher_Init(int type, char *key, size_t keylen);`
Stvara kontekst i inicijalizira algoritam sa priloženim ključem. Kod algoritama koji imaju određenu veličinu ključa, zadnji parametar se ignorira. U slučaju greške funkcija vraća NULL, inače vraća pokazivač na novi kontekst.

Encrypt()

Prototip: `CYFER_StreamCipher_Encrypt(CYFER_STREAM_CIPHER_CTX *ctx, char *input, char *output, size_t size);`
Kriptira blok podataka određene veličine (minimalno 0 okteta).

Decrypt()

Prototip: `CYFER_StreamCipher_Decrypt(CYFER_STREAM_CIPHER_CTX *ctx, char *input, char *output, size_t size);`
Dekriptira blok podataka određene veličine (minimalno 0 okteta).

StreamCipher_Finish()

Prototip: `CYFER_StreamCipher_Finish(CYFER_STREAM_CIPHER_CTX *ctx);`
Završava rad algoritama i uništava kontekst.

A.4 – Funkcije za rad sa asimetričnim algoritmima

Potrebna header datoteka: `pk.h`

Get_Supported()

Prototip: `CYFER_Pk_t *CYFER_Pk_Get_Supported();`
Vraća pokazivač na niz struktura koje opisuju podržane algoritme.

Select()

Prototip: `int CYFER_Pk_Select(char *name, bool *enc, bool *sig);`
Odabire algoram pod nazivom 'name' – 'enc' je istinito ako se algoritam može koristiti za kriptiranje, 'sig' je istinito ako se algoritam može koristiti za potpisivanje poruka.

Init()

Prototip: `CYFER_PK_CTX *CYFER_Pk_Init(int type);`
Stvara i inicijalizira kontekst za specificirani algoritam. U slučaju greške (nepodržani algoritam) vraća vrijednost NULL, inače vraća pokazivač na novi kontekst.

Generate_Key()

Prototip: `CYFER_Pk_Generate_Key(CYFER_PK_CTX *ctx, size_t size);`
Generira privatni i javni ključ specificirane duljine (u bitovima).

KeySize()

Prototip: `CYFER_Pk_KeySize(CYFER_PK_CTX *ctx, size_t *private, size_t *public);`
Vraća potrebne veličine spremnika za privatni i javni ključ. Vrijednosti su izražene u oktetima.

Export_Key()

Prototip: `CYFER_Pk_Export_Key(CYFER_PK_CTX *ctx, char *private, char *public);`
Izvozi javni i privatni ključ. Pretpostavlja se da spremnici imaju dovoljno mjesta za ključeve. U slučaju da se želi izvesti samo jedan ključ, za drugi se može kao spremnik postaviti `NULL` vrijednost.

Import_Key()

Prototip: `CYFER_Pk_Import_Key(CYFER_PK_CTX *ctx, char *private, size_t privlen, char *public, size_t publen);`
Uvozi javni i privatni ključ. U slučaju da se želi uvesti samo jedan ključ, za drugi se može kao spremnik postaviti `NULL` vrijednost. Funkciji je potrebno predati i duljine spremnika ključeva (ne duljine samih ključeva), koje se izražavaju u oktetima.

Size()

Prototip: `CYFER_Pk_Size(CYFER_PK_CTX *ctx, size_t *textlen, size_t *cipherlen);`
Vraća duljine blokova jasnog i kriptiranog teksta koje koristi algoritam. Vrijednosti su izražene u oktetima.

Encrypt()

Prototip: `CYFER_Pk_Encrypt(CYFER_PK_CTX *ctx, char *text, char *cipher);`
Kriptira jasni tekst javnim ključem. Veličine spremnika za jasni i kriptirani tekst moraju odgovarati veličinama koje je vratila funkcija Size().

Decrypt()

Prototip: `CYFER_Pk_Decrypt(CYFER_PK_CTX *ctx, char *cipher, char *text);`
Dekriptira jasni tekst privatnim ključem. Veličine spremnika za jasni i kriptirani tekst moraju odgovarati veličinama koje je vratila funkcija Size().

Sign()

Prototip: `CYFER_Pk_Sign(CYFER_PK_CTX *ctx, char *message, char *signature);`
Potpisuje poruku privatnim ključem. Veličine spremnika za poruku (jasni tekst) i potpis (kriptirani tekst) moraju odgovarati veličinama koje je vratila funkcija Size().

Verify()

Prototip: `bool CYFER_Pk_Verify(CYFER_PK_CTX *ctx, char *signature, char *message);`
Provjerava potpis poruke koristeći javni ključ. Spremnik poruke mora sadržavati poruku koja se provjerava. Veličine spremnika moraju odgovarati veličinama koje je vratila funkcija Size(). Funkcija vraća vrijednost `true` ako je potpis valjan, odnosno `false` ako nije.

Finish()

Prototip: `CYFER_Pk_Finish(CYFER_PK_CTX *ctx);`
Završava sa korištenjem algoritma i uništava kontekst.

A.5 – Funkcije za razmjenu sjedničkih ključeva

Potrebna header datoteka: `keyex.h`

Get_Supported()
Prototip: `CYFER_Pk_t *CYFER_KeyEx_Get_Supported();`
Vraća pokazivač na niz struktura koje opisuju podržane algoritme.

Select()
Prototip: `int CYFER_KeyEx_Select(char *name);`
Odabire algoram pod nazivom 'name', vraća -1 ako algoritam nije podržan.

KeyEx_Init()
Prototip: `CYFER_KEYEX_CTX *CYFER_KeyEx_Init(int type);`
Inicijalizira kontekst za specificirani algoritam. U slučaju greške (nepodržani algoritam) vraća vrijednost NULL, inače vraća pokazivač na novi kontekst.

Generate_Key()
Prototip: `CYFER_KeyEx_Generate_Key(CYFER_KEYEX_CTX *ctx);`
Generira privatni i javni ključ.

KeySize()
Prototip: `CYFER_KeyEx_KeySize(CYFER_KEYEX_CTX *ctx, size_t *shared, size_t *public);`
Vraća potrebnu veličinu spremnika za javni ključ, te izvornu duljinu generiranog sjedničkog ključa. Vrijednosti su izražene u oktetima.

Public_Key()
Prototip: `CYFER_KeyEx_Public_Key(CYFER_KEYEX_CTX *ctx, char *public);`
Izvozi javni ključ. Pretpostavlja se da spremnik ima dovoljno mjesta za ključ.

Compute_Key()
Prototip: `CYFER_KeyEx_Compute_Key(CYFER_KEYEX_CTX *ctx, char *other, size_t size);`
Uvozi javni ključ druge strane i računa sjednički ključ.

Shared_Key()
Prototip: `CYFER_KeyEx_Public_Key(CYFER_KEYEX_CTX *ctx, char *shared, size_t size);`
Izvozi sjednički ključ. Moguće je zatražiti bilo koju duljinu ključa – generirani ključ kopira se u spremnik onoliko puta koliko ima mjesta (ako je tražena duljina 100 okteta a sjednički ključ je velik 40 okteta, kopirat će se 2.5 puta). Ako je tražena duljina manja od sjedničkog ključa, kopirat će se samo dio ključa.

KeyEx_Finish()
Prototip: `CYFER_KeyEx_Finish(CYFER_KEYEX_CTX *ctx);`
Završava sa korištenjem algoritma i uništava kontekst.

Dodatak B – Korištenje Cyfer biblioteke u vlastitim programima

Biblioteka Cyfer na većini podržanih operacijskih sustava dolazi u obliku dinamičke biblioteke (*shared library*, *dynamically linked library*) i statičke biblioteke (*static library*).

Prednost dinamičke biblioteke je povezivanje s programom prilikom samog izvođenja programa, što omogućuje jednostavnu zamjenu biblioteke bez ponovnog prevođenja programa, osiguravanje da svi programi na sustavu koriste istu inačicu biblioteke, kao i smanjenje zauzeća diskovnog prostora (jer svi programi koriste istu kopiju biblioteke). Nedostatak ovakvog pristupa je zavisnost programa o vanjskim datotekama (tj. o biblioteci). Osim toga, neki stariji operacijski sustavi nemaju mogućnost korišetnja dinamičkih biblioteka.

Prednost korištenja statičke inačice biblioteke je smanjenje zavisnosti (sva potrebna funkcionalnost ugrađuje se u program prilikom prevođenja).

Korištenje biblioteke u Unix operacijskim sustavima jednako je kod dinamičke i statičke biblioteke (uz izuzetak da dinamička biblioteka mora postojati na sustavu gdje se sam program izvodi). Primjer prevođenja programa GNU C prevodiocem uz korištenje dinamičke biblioteke:

```
gcc -o test test.c -lcyfer
```

Argument `-lcyfer` kazuje prevodiocu da je potrebno uključiti i Cyfer biblioteku. Kod korištenja statičke biblioteke, potrebno je samo prevodiocu reći da koristi statičko povezivanje, argumentom `-static`.

Prilikom prevođenja programa u Windows operacijskim sustavima, povezivanje sa potrebnim bibliotekama provodi se navođenjem biblioteke u "*Additional Linker Dependencies*" polje u dijalogu svojstava ("*Properties*") programa.

Za korištenje Cyfer biblioteke u tom polju potrebno je staviti: "`libcyfer.lib libgmp.lib`", čime se kazuje prevodiocu da je potrebno uključiti Cyfer i GMP biblioteke. Prilikom korištenja dinamičke biblioteke koriste se *stub* biblioteke koje će prilikom izvođenja programa učitati dinamičku biblioteku (koja ima nastavak `.dll`). Kod korištenja statičke biblioteke, navodi se sama biblioteka.

Imena statičkih biblioteka i *stub* biblioteka su jednaka, pa je potrebno paziti da ih se ne zamijeni. Same datoteke nalaze se u različitim direktorijima.

Literatura

[1] L. Budin, "Operacijski sustavi 2 - bilješke s predavanja", 2004
[2] Fakultet Elektrotehnike i Računarstva, "Radovi studenata iz područja računalne sigurnosti", 2004, http://sigurnost.zemris.fer.hr/
[3] A. Menzes, P. van Oorschot, S. Vanstone, "Handbook of Applied Cryptography", *CRC Press*,1996
[4] B. Kalinski, "RFC1319 - The MD2 Message-Digest Algorithm", 1992, http://www.faqs.org/rfcs/rfc1319.html
[5] R. Rivest, "RFC1320 - The MD4 Message-Digest Algorithm", 1992, http://www.faqs.org/rfcs/rfc1320.html
[6] R. Rivest, "RFC1321 - The MD5 Message-Digest Algorithm", 1992, http://www.faqs.org/rfcs/rfc1321.html
[7] National Institute for Standards and Technology, "FIPS 180-2 - Secure Hash Standard", 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
[8] H. Dobbertin, A. Bosselaers, B. Preenel, "RIPEMD-160 - A strengthened version of RIPEMD", *Fast Software Encryption, LNCS1039, pp.71-82.*, 1996
[9] R. C. Merkle, "A Fast Software One-Way Hash Function", *Journal of Cryptology 3:1, pp 43--58.*, 1990
[10] L. P. Deutsch, J.L. Gailly, "RFC 1950 - ZLIB Compressed Data Format Specification version 3.3", 1996, http://www.faqs.org/rfcs/rfc1950.html
[11] National Institute of Standards and Technology, "FIPS 46-3 - Data Encryption Standard", 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
[12] National Institute for Standards and Technology, "FIPS 197 - Advanced Encryption Standard", 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[13] X. Lai, "On the Design and Security of Block Ciphers", Ph.D. thesis, *Swiss Federal Institute of Technology*, 1992
[14] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)", *Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), pp. 191-204.*, 1994
[15] Lars R. Knudsen, "DEAL - A 128-bit Block Cipher", Technical Report 151, *Department of Informatics, University of Bergen*, 1998
[16] R. Rivest, "RFC 2268 - A Description of the RC2 Encryption Algorithm", 1998, http://www.faqs.org/rfcs/rfc2268.html
[17] K. Kaukonen, R. Thayer, "A Stream Cipher Encryption Algorithm "Arcfour"", 1999, http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt
[18] R. Rivest, "The RC5 Encryption Algorithm", *Fast Software Encryption, LNCS1008, pp 86--96.*, 1995
[19] R. Rivest, M. J. B. Robshaw, R. Sidney, Y. L. Lin, "The RC6 Block Cipher", 1998, http://theory.lcs.mit.edu/~rivest/rc6.ps
[20] J. Daemen, "A New Approach to Block Cipher Design", *Proceedings of the Cambridge Workshop on Cryptography 1993*, 1994
[21] P. Smith, M. Lennon, "LUC: A New Public Key System", *Ninth IFIP Symposium on Computer Security, pp 103-117.*, 1993