Programsko sučelje za kriptografske algoritme prilagođene ugrađenim sustavima
Tehnička dokumentacija
Verzija 2.0

Studentski tim: Iva Boras
Ana Geto
Laura Majer
Marin Njirić
Patrik Okanović
Ivan Rissi
Stipe Šuto

Nastavnik: prof. dr. sc. Marin Golub

Sadržaj

1. Uvod 3
1.1 O kriptografskim algoritmima prilagođenim ugrađenim sustavima 3
1.2 Rezultati projekta 3
1.2.1 Program 3
1.2.2 Prezentacija 4
1.2.3 Web stranica 4

2. Analiza algoritama 5
2.1 Elephant 5
2.2 GIFT-COFB 6
2.3 GRAIN-128AEAD 9
2.4 HyENA 11
2.5 LOCUS-LOTUS 13
2.6 MIXFEED 15
2.7 ACE 17
2.8 GIMLI 19
2.9 ORANGE 21
2.10 Romulus 23
2.11 Spook 29
2.12 PHOTON-Beetle 31
2.13 Subterranean 2.0 34
2.14 Xoodyak 36

3. Tehničke značajke 39
3.1 Prevođenje algoritama 39
3.2 Izrada .dll datoteka 41

4. Upute za korištenje 46

5. Literatura 51

Tehnička dokumentacija

1. Uvod

1.1 O kriptografskim algoritmima prilagođenim ugrađenim sustavima

Autentifikacijska enkripcija je oblik enkripcije koji, osim pružanja povjerljivosti poruke, pruža način da se provjeri i integritet poruke, odnosno provjera je li sadržaj poruke tijekom transporta od izvorišta do odredišta ostao nepromijenjen. Autentifikacijska enkripcija sa povezanim podacima (eng. *Authenticated Encryption with Associated Data* – AEAD) omogućuje provjeru integriteta i autentičosti dodatnih povezanih podataka (eng. *Associated Data* – AD) koji nisu kriptirani [15].
Razvojem područja poput distribuiranih kontrolnih sustava te interneta stvari (engl. Internet of things) u kojima su uređaji najčešće međusobno bežično povezani raste potreba za zaštitom njihove komunikacije. S obzirom da je većina razvijenih kriptografskih algoritama prilagođena računalima veće procesorske snage, dolazi do potrebe za razvojem novih algoritama. NIST je pokrenuo postupak za traženje, procjenu i standardizaciju kriptografskih algoritama koji su prikladni za upotrebu u ograničenim okruženjima u kojima performanse trenutnih kriptografskih standarda NIST nisu prihvatljive.
Prema natječaju potrebno je razviti funkciju s 4 ulaza i 1 izlazom tipa *bytestring.* Ulazi varijabilne duljine su: *plaintext i data*. Ulazi fiksne duljine su: *nonce* (>= 96 bita*) i key* (>= 128 bita*).* Izlaz varijabilne duljine je *ciphertext*. *Hash* funkciju nije bilo potrebno razviti za prijavu na NISTov natječaj.

NIST uz natječaj objavljuje ove dvije datoteke:

1. genkat_aead.c → izvorni tekst programa za stvaranje ispitnih vektora
2. crypto_aead.h → zaglavlje koje specificira metode koje se moraju programski ostvariti (crypto_aead_encrypt i crypto_aead_decrypt)

Obje datoteke moraju bez izmjena biti uključene u mape algoritama te se programski kod natjecatelja mora referirati na njih. Valjana prijava na natječaj mora sadržavati:

1. api.h → zaglavlje koje definira duljinu ključa i ostale parametre. Podaci iz api.h koriste se u stvaranju ispitnih testnih vektora.
2. encrypt.c (preporučeno ime, ovo može biti podijeljeno u više .c datoteka) → izvorni tekst programa koji sadrži programsko ostvarenje metoda iz crypto_aead.h

1.2 Rezultati projekta

1.2.1 Program

Glavni rezultat ovog projekta je programsko sučelje koje spaja odabrane algoritme za autentifikacijsku kriptografiju prijavljene na NISTov natječaj. Odabrani su sljedeći algoritmi:

- ACE
- Elephant
- GIFT-COFB
- Gimli
- Grain-128AEAD
- HYENA
- LOTUS-AEAD i LOCUS-AEAD
- Mixfeed
- ORANGE
- PHOTON-Beetle

- Romulus
- Spook
- Subterranean 2.0
- Xoodyak

Član tima zadužen za ostvarenje programa je Stipe Šuto.

1.2.2 Prezentacija

Kao dio projekta izrađena je prezentacija koja ukratko predstavlja odabrane algoritme i te izrađeno programsko sučelje. Za prezentaciju su zadužene Ana Geto te Iva Boras.

1.2.3 Web stranica

Na web stranici omogućeno je preuzimanje prezentacije, tehničke doumentacije i programskog sučelja. Član zadužen za izradu web stranice je Ivan Rissi.

2. Analiza algoritama

Svi odabrani algoritmi zadovoljavaju uvjete NISTovog natječaja te samim time svojim implementacijama nužno je da budu AEAD algoritmi. Odnosno, moraju sadržavati funkciju s 4 ulaza i 1 izlazom tipa bytestring. Ulazi varijabile duljine su : *plaintext i data.* Ulazi fiksne duljine: *nonce (*>= 96 bita), *key* (>= 128 bita). Izlaz varijabilne duljine: *ciphertext.*
Neobavezni dio implementacije algoritama za natječaj je funkcija za izračunavanje sažetka (*hash* funkcija). *Hash* funkcija prima ulaz varijabilne duljine *message*, a vraća izlaz fiksne duljine *hash value*.

2.1 Elephant

Specifikacije

veličina ključa primarnog algoritma	128 bita
alternativne veličine ključeva	-
veličina bloka	160 bita
metoda nadopune zadnjeg bloka	0*
implementiran algoritam HASH	NE

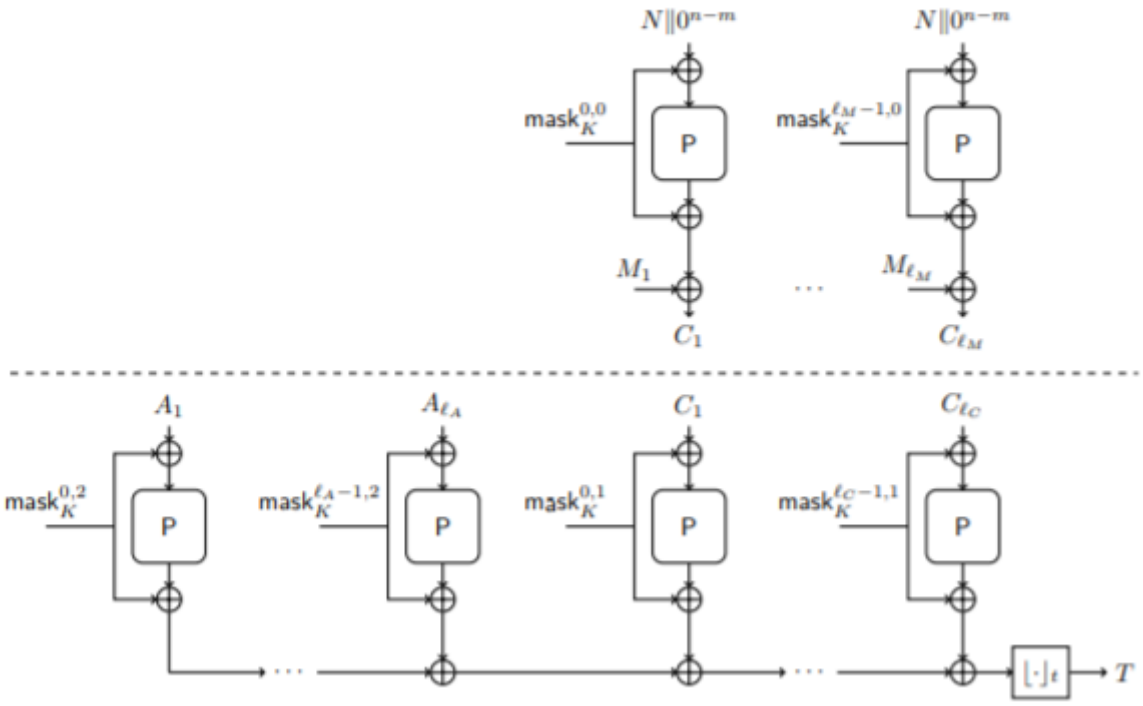
Namjena i glavna ideja: kriptiranje se izvodi pomoću autentifikacije poruke varijantom Wegman-Carter-Shoup MAC funkcije. Ona iznutra koristi kriptografsku permutaciju maskiranu pomoću LFSR(eng. Linear-feedback shift register). Shema Elephant algoritama sastoji se od tri instance: Dumbo, Jumbo i Delirium.

Osnovna svojstva algoritma: omogućen paralelizam, jednostavan za implementaciju zbog korištenja LFSR za maskiranje. Efikasan zbog elegantnih odluka kako točno treba maskiranje biti izvedeno. Zbog paralelnosti, nema potrebe za stvaranje Elephant-a sa velikom permutacijom – može se koristiti 160-bitna permutacija i dalje dosegnuti sve ciljeve sigurnosti.

Opis algoritma:

Parametri: ključ duljine 128 bitova, *nonce* duljine 96 bitova
U algoritmu se koristi modificirana spužvasta funkcija. Ona permutaciju s fiksnom duljinom ulaza i izlaza omata u funkciju koja prima ulaz proizvoljne konačne duljine i daje izlaz proizvoljne konačne duljine. Za spužvastu funkciju bitna je širina *bitrate*-a unutar stanja koja se označava s r jer se s prvih r bitova obavljaju XOR operacije te se oni prosljeđuju onoj funkciji koja se „omotava“ spužvastom funkcijom. Spužva se sastoji od dvije faze - faze upijanja i faze cijeđenja.
Faze algoritma:

- **Kriptiranje –** dobiva ključ $K \in \{0, 1\}^k$ kao ulaz, *nonce* $N \in \{0, 1\}^m$, oznaku $T \in \{0, 1\}^k$, poruku $M \in \{0, 1\}^*$, a kao izlaz kriptirani tekst $C \in \{0, 1\}^{|M|}$
- **Dekriptiranje -** dobiva ključ $K \in \{0, 1\}^k$ kao ulaz, *nonce* $N \in \{0, 1\}^m$, kriptirani tekst $C \in \{0, 1\}^*$ i $T \in \{0, 1\}^k$, a kao izlaz poruku $M \in \{0, 1\}^{|M|}$ ako je oznaka točna, inače znak $\perp$
- **160-bitna permutacija i LFSR –** ovdje se koristi spužvasta funkcija i 160-bitno maskiranje. Ove komponente se koriste u instanci Dumbo. Djeluje na 160-bitnom ulazu X koristeći XOR te metode ICounter(i), sBoxLayer(X) i pLayer(X)



Slika 1. Prikaz Elephant kriptiranja

Implementacija: Za varijante Elephant-a orijentirane prema sklopovlju (Dumbo i Jumbo), visoka razina paralelizma omogućava jednostavno povećanje propusnosti. Programska implementacija Elephant-a (Delirium) također može imati koristi od paralelizma. Ako su dostupne više jezgri odjednom, više blokova mogu biti obrađeni odjednom, ali ovo je korisno samo za dugačke poruke.

Sigurnost: Dumbo – otpornost na sve napade: 2^{160}
Jumbo – otpornost na sve napade: 2^{176}
Delirium – otpornost na sve napade: 2^{200}

2.2 GIFT-COFB

Specifikacije

veličina ključa primarnog algoritma	128 bita
alternativne veličine ključeva	-
veličina bloka	128 bita
metoda nadopune zadnjeg bloka	Slika 2.
implementiran algoritam HASH	NE

Namjena i glavna ideja: GIFT-COFB koristi blokovski kriptografski algoritam COFB (Combined FeedBack) baziran na AEAD algoritmu (*Authenticated Encryption with Associated Data*) sa optimiziranim blokovskim kriptografskim algoritmom GIFT-128. Kombiniranjem povratnih informacija iz izlaza (*outputa*) blokovskog kriptografskog algoritma s unaprijeđenom XEX metodom (XOR–kriptiranje–XOR) postignuto je minimiziranje količine maskiranja za faktor 2.

Osnovna svojstva algoritma:
Potreban je samo jedan poziv blokovskog kriptografskog algoritma za jedan ulazni blok.
Nije potrebno raditi inverz u postupku dekriptiranja.
Mali skup stanja, u slučaju kada algoritam kriptiranja bloka prima n-bitni blok i k-bitni ključ, koristi samo 1.5n+k bitova.

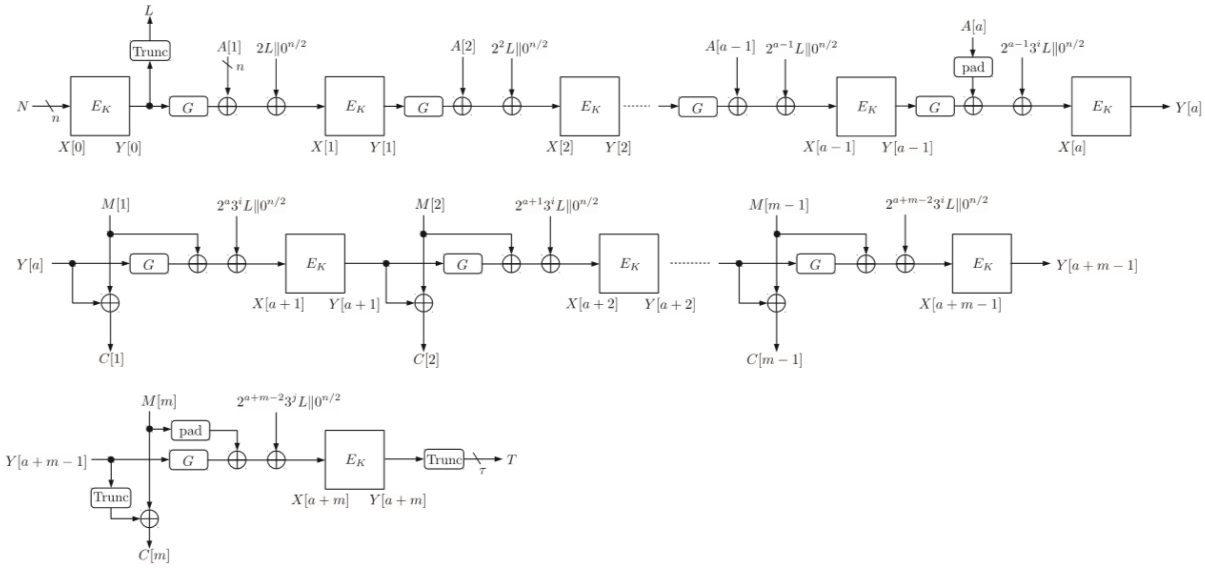
Opis algoritma:

Parametri: ključ duljine 128 bitova, *nonce* duljine 128 bitova
Blokovski kripto algoritam GIFT-128 koristi 4 identične *Round* funkcije. Ona se sastoji od inicijalizacije i 3 koraka: *SubCells, PermBits*, i *AddRoundKey*. *Plaintext* veličine 128 bita se na početku učita u kriptirano stanje S, koje će biti podijeljeno na četiri 32-bitna segmenta (S0 -S3).
Subcells: pomoću zadanih operacija ažuriraju se stanja (S0-S3).
PermBits: permutacije 32-bitnih stanja.
AddRoundKey: dodavanje *round* ključa i konstante.
Koraci *COFB* dijela AEAD algoritma (*encryption*):

Temelj enkripcijskog algoritma je n-bitni blokovski kripto algoritam E_k(K je vrijednost ključa). Prvo se odrede početni ulazni blokovi i konstante koji će nam trebati za enkripciju. Koristimo blokovski kripto algoritam GIFT-128 opisan u gornjem dijelu. Koriste se *Padding* funkcija i povratna funkcija.
Padding funkcija funkcionira na način prikazan na slici 3., a povratna funkcija se koristi za generiranje kriptiranog stanja pomoću nasumično izabranih parametara ($Y \in \{0,1\}^n$, $X \in Y$). Detaljan način je opisan u tehničkoj dokumentaciji.
Algoritam prima A i M (pridruženi podaci i poruka), a rezultira C i T (kriptirani tekst i oznaka) tako da je duljina C jednaka duljini M, a duljina T jednaka n.

$$\mathsf{Pad}(x) = \begin{cases} x & \text{if } x \neq \epsilon \text{ and } |x| \bmod n = 0 \\ x \| 10^{(n-(|x| \bmod n)-1)} & \text{otherwise.} \end{cases}$$

Slika 2. Padding funkcija algoritma



Slika 3. prikaz tijeka kriptiranja unutar AEAD algoritma

Sklopovski zahtjevi: COFB način rada dizajniran je s *rate* vrijednošću 1, odnosno svaki blok poruka se obrađuje samo jednom. To uvjetuje dobru propusnost i manju potrošnju energije. No da bi se omogućio ovakav način rada potrebno je dodatno 64-bitno stanje što zahtjeva dodatan 64-bitni registar na sklopovlju koje ga implementira.

Dekripcija: simetrična kriptiranju, što znači da su koraci dekriptiranja ekvivalentni koracima opisanima u algoritmu za kriptiranje, samo se korak samog dekriptiranja razlikuje u parametrima. Dekripcijski algoritam prima (N, A, C, T), a rezultira M (porukom) ili ⊥ (odbijanje).

Sigurnost:

Construction	State Size(bits)	IND-CPA(bits)	INT-CTXT(bits)
GIFT-COFB	192 (excluding the key state)	64	58

Slika 4. Sigurnost IND-CPA i INT-CTXT, poštujući *nonce*

2.3 GRAIN-128AEAD

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	64 bita
metoda nadopune zadnjeg bloka	za poruku $m_{0...}m_{L-1}$ duljine L postavlja se m_L=1 i još sedam nula
implementiran algoritam HASH	NE

Namjena i glavna ideja: *Authenticated Encryption with Associated Data* algoritam (u daljnjem tekstu naznačen s AEAD). Svi Grain slijedni kriptografski algoritmi također omogućuju povećanje propusnosti dodavanjem kopije uključenih Booleovih funkcija.

Osnovna svojstva algoritma: Ne implementira HASH funkciju. Korištenje maske omogućava veću fleksibilnost protokola. Ne sprema ključ u postojanu podatkovnu memoriju što omogućuje da se ključ može ažurirati na uređaju (iskoristiv na većem rasponu uređaja).

Opis algoritma:

Parametri: ključ duljine 128 bitova, *nonce* duljine 96 bitova

Grain-128AEAD se sastoji od dva glavna sastavna bloka. Prvi je pred izlazni generator koji se sastoji od *Linear Feedback Shift Register* (LFSR), *Non-linear Feedback Shift Register* (NFSR) i pred izlazne funkcije. On generira pseudoslučajne bitove, koji se koriste za oznaku autentičnosti. Drugi je generator autentikator koji se sastoji od posmačnog spremnika, koji sadrži najnovija 64 neparna bita iz pred izlazne funkcije, te akumulatora.

Koraci AEAD algoritma (*encryption*):

- **učitavanje ključa** i *nonce*-a (jednokratne javne riječi),
- **inicijalizacija stanja** pred izlaznog generatora i registara generatora autentikatora pomoću ključa i *nonce*-a
- **procesuiranje *Additional data* –** definiramo AEAD masku koja specifira koje bitove treba kriptirati
- **kriptiranje poruke** – Nakon inicijalizacije pred izlaznog generatora, pre-output se koristi za generiranje nizovnog ključa z_i potrebnog za kriptiranje. Poruka m se kriptira tako da: $C_i = m_i \oplus z_i$, $(0 \leq i <$ duljina poruke m$)$
- kriptiranje i MAC generiranje s NIST API-jem prikazano je na slici 5.

Algorithm 1: AEAD Encrypt with NIST API

Input: ad, $adlen$, m, $mlen$, k, $nonce$
Output: c

Initialize generator with k and $nonce$
Construct $m' = (\text{Encode}(adlen) \| ad \| m \| 0\text{x}80)$
Let
  $M =$ bit length of $\text{Encode}(adlen) \| ad$
  $d_i = 0, \quad (0 \leq i \leq M-1)$
  $d_i = 1, \quad (M \leq i \leq M + mlen - 1)$
Encrypt using $c'_i = m'_i \oplus z_i d_i, \quad (0 \leq i \leq M + mlen - 1)$
Authenticate using z'_i and generate $A_{M+mlen+1}$
$c = (c'_M, c'_{M+1}, \ldots, c'_{M+mlen-1}) \| A_{M+mlen+1}$
return 0

Slika 5. prikaz tijeka kriptiranja unutar AEAD algoritma

Sklopovski zahtjevi: može se konstruirati pomoću primitivnih hardverskih blokova, poput NAND-a, XOR-a i takozvanog *flip flop* sklopovlja.

Dekripcija: simetrična kriptiranju, što znači da su koraci dekriptiranja ekvivalentni koracima opisanima u algoritmu kriptiranja, samo se korak samog dekriptiranja razlikuje u parametrima. Detaljan prikaz na slici 6.

Algorithm 2: AEAD Decrypt with NIST API

Input: ad, $adlen$, c, $clen$, k, $nonce$
Output: m

Initialize generator with k and $nonce$
Construct $c' = (\text{Encode}(adlen) \| ad \| c_0, \ldots, c_{clen-65} \| 0\text{x}80)$
Let
  $M =$ bit length of $\text{Encode}(adlen) \| ad$
  $mlen = clen - 64$
  $d_i = 0, \quad (0 \leq i \leq M-1)$
  $d_i = 1, \quad (M \leq i \leq M + mlen - 1)$
Decrypt using $m'_i = c'_i \oplus z_i d_i, \quad (0 \leq i \leq M + mlen - 1)$
Authenticate using z'_i and generate $A_{M+mlen+1}$
Set $m = m'_M, \ldots, m'_{M+mlen-1}$
if $(c_{clen-64}, \ldots, c_{clen-1}) == A_{M+mlen+1}$
  return 0
else
  return -1

Slika 6. prikaz tijeka kriptiranja unutar AEAD algoritma

Sigurnost:

AEAD – otpornost na sve napade: 2^{128}

2.4 HyENA

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	128 bitova
metoda nadopune zadnjeg bloka	$Pad(X) = \begin{cases} X & ako\ \|X\|\ mod\ n = 0 \\ 0^{n-\|X\|-1}\|1\|\|X & inače \end{cases}$
implementiran algoritam HASH	NE

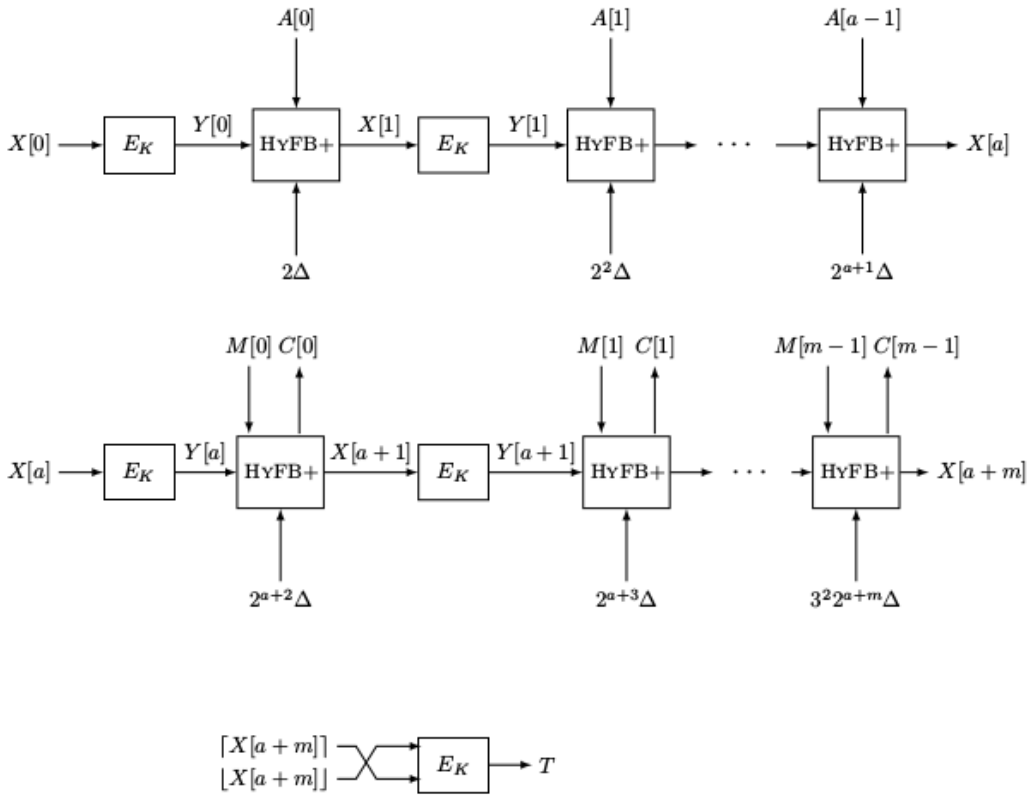
Namjena i glavna ideja: HyENA (engl. Hybrid feedback-based ENcryption with Authentication) implementira AEAD (engl. *nonce-based authenticated encryption with associated data).* Uobičajeno enkriptiranje koristi jednu od sljedećih metoda kripiranja: PFB *(engl. plaintext feedback),* CFB *(engl. ciphertext feedback), ili* OFB *(engl. output feedback)*. HyENA je hibrid povratnih poruka i blokova kriptiranog teksta. Ulaz blokova kriptiranog teksta je djelomično CFB te PFB. HyENA je osmišljena za „siromašno sklopovlje". Glavna ideja jest minimizirati veličinu stanja iznad stanja kriptiranog bloka (uključujući *key schedule*), te reducirati broj XOR-anja. Neka od zanimljivih svojstava HyENA-e su *single-pass* (jedan kriptirani blok po bloku podataka), *inverse-free* (nema potrebe za dekripcijiom blocka *ciphera*) te iznimno mala veličina stanja, otprilike 1.5n + k za kriptirani blok sa n-bitnim blokom i k-bitnim ključem. HyENA se instancira sa kriptiranim blokom GIFT-128.

Osnovna svojstva:

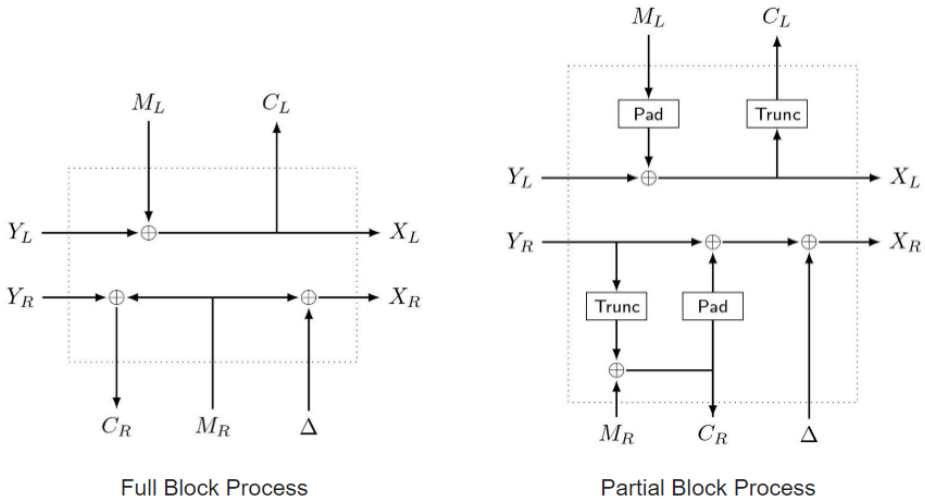
- **Optimalnost:** HyENA zahtjeva (a + m + 1) poziva kriptiranih blokova kako bi se obradio a blok s odgovarajućim podacima i m blok poruka. To je optimalni broj nelinearnih primitivnih poziva za bilo koji slučajni broj koji se jednom koristi (engl. *nonce)* bazirani autentifikacijski algoritam.
- **Inverse-free:** Ni enkriptiranje, ni verificirano dekriptiranje ne zahtjevaju poziv na odgovarajući kriptirani blok. Time je znatno smanjen *hardware footprint*, posebice u implementacijama koje koriste i enkriptiranje i dekriptiranje.
- **Niski broj stanja**: HyENA poput COFB-a zahtjeva stanje veličine 3n/2 bita. Za bilo koju *nonce* baziranu konstrukciju, može se dokazati da to predstavlja optimalnu veličinu stanja.
- **Mali broj XOR-anja:** Za ostvarenje optimalnosti, *inverse-free authenticated ciphers with low state*, moguća opcija jest koristiti kombinirani povratni pristup gdje (i) prethodni izlazni kriptirani blok je XOR-an sa tekstom kako bi se dobio kriptirani tekst, i (ii) sljedeći kriptirani ulazni blok se definira XORanjem teksta s nekom linearnom funkcijom prethodnog kriptiranog izlaznog bloka. Ova tehnika koristi se u autentifikacijskojm enkriptiranju, COFB-u. Vidljivo je da takva kombinacija povratnih funkcija zahtjeva najmanje 2n-bita XOR operacija (kada se obrađuje n-bitni podatak), uz neke dodatne XORove vezane za linearnu funkciju prethodno spomenutu. Unatoč tome HyENA koristi hibridnu povratnu vrijednost ili HyFB, gdje se ulaz kriptiranog bloka definira djelomice pomoću povratne vrijednosti kriptiranog teksta i djelomice pomoću običnog teksta. To smanjuje broj XOR operacija na n.

Opis algoritma: Koristi se kriptirani blok sa n = 128 i k = 128.
Parametri: k-bitni enkriptirajući ključ, r-bitni *nonce* N sa r = 96 (autentificiran, ali ne enkriptiran) i proizvoljno dugačak podatak A (autentificiran, ali ne enkriptiran), i proizvoljno dugačku poruku M (autentificiranu i enkriptiranu)
Izlaz : kriptirani tekst *C* za koji vrijedi |C| = |M| i t-bitni tag T s t = 128.

Dekriptirajući algoritam prima K, A, N, C i T kao ulaz, a vraća M ako C odgovara T, inače odbacuje. Slika 7. prikazuje prethodno opisano. Povratna funkcija obrađuje pune ili parcijalne blokove podataka na drugačiji način. Povratna funkcija prikazana je na slici 8.



Slika 7. Ilustracija rada algoritma



Slika 8. Povratna funkcija

Sigurnost: Tablica prikazuje granične vrijednost podataka i vremena potrebnih da bi se vjerojatnost uspješnosti napada približila 1

Mode	Privacy		Authenticity	
	Time	Data (in bytes)	Time	Data (in bytes)
HyENA	2^{128}	2^{64}	2^{128}	2^{58}

Slika 9. Tablica sigurnosti HyENA

2.5 LOCUS-LOTUS

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	64 bita
metoda nadopune zadnjeg bloka	XEX duljina zadnjeg bloka, XOR izlaz s porukom zadnjeg bloka
implementiran algoritam HASH	DA

Namjena i glavna ideja: dva nova načina rada, nazvana LOTUS-AEAD i LOCUS-AEAD bazirana na AEAD algoritmu (*Authenticated Encryption with Associated Data*) sa optimiziranim blokovskim kriptografskim algoritmom TweGIFT-64.

Osnovna svojstva algoritma:

Opis algoritma:

Parametri: ključ duljine 128 bitova, broj koji se koristi samo jednom(dalje u dokumentu *nonce*) duljine 128 bitova
TweGIFT-64 se sastoji od 28 rundi, a svaka runda se sastoji od sljedećih operacija: *Subcells* (S-kutije koje primjenjuje na kriptirana stanja), PermBits (permutacije), *AddRoundKey* (dodavanje 32-bitnog *round* ključa, izvedenog iz prim. kljuca, kriptiranom stanju), *AddRoundConstant, AddTweak* (dodavanje konstante i varijable *Tweak*)

1) Koraci LOTUS algoritma (kriptiranje):

-podaci se podijele u 2n bitne diblokove i obrađuju se na sličan način kao OTR.
- dva uzastopna kriptiranja i u gornjem i u donjem sloju.
- za obradu dibloka ključ se prvo ažurira množenjem s α
-obrada dibloka: Za gornji sloj se koristi *tweak* 0100, a za donji 0101. Time se postiže odvajanje domena.

Vrijedi za sve osim zadnjeg dibloka gdje se koristi njegova duljina.
-Tag se generira tako da se koristi XEX (xor–encrypt–xor).
-detaljan opis na slici

2) Koraci LOCUS algoritma (*kriptiranje*):
-podaci se dijele u n-bitne blokove i obrađuju se na sličan način kao OCB
-prvo se maskira blok, zatim kriptira koristeći *tweakable* blokovski kriptirajući algoritam i zatim se opet maskira
-zadnji blok, umjesto da koristi XEX na cijelom bloku, koristi se na duljini i zatim XOR sa zadnjim blokom
-na isti način kao i LOTUS algoritam generira se Tag i potom ažurira ključ

Algorithm 3 The encryption and verification-decryption algorithms of LOCUS-AEAD. The subroutine proc_ad and proc_tag are identical to the one used in LOTUS-AEAD.

function LOCUS-AEAD-$\widetilde{\mathsf{E}}$.enc(K, N, A, M)
  $C \leftarrow \perp$, $W_{\oplus} \leftarrow 0$, $V_{\oplus} \leftarrow 0$
  $(K_N, \Delta_N) \leftarrow \mathsf{init}(K, N)$
  if $|A| \neq 0$ **then**
    $(K_N, V_{\oplus}) \leftarrow \mathsf{proc\_ad}(K_N, \Delta_N, A)$
  if $|M| \neq 0$ **then**
    $(K_N, W_{\oplus}, C) \leftarrow \mathsf{proc\_pt}(K_N, \Delta_N, M)$
  $T \leftarrow \mathsf{proc\_tg}(K_N, \Delta_N, V_{\oplus}, W_{\oplus})$
  return (C, T)

function $\mathsf{proc\_pt}(K_N, \Delta_N, M)$
  $L \leftarrow K_N$
  $(M_{m-1}, \ldots, M_0) \xleftarrow{n} M$
  for $j = 0$ **to** $m - 2$ **do**
    $X \leftarrow M_j \ominus \Delta_N$
    $L \leftarrow L \odot \alpha$
    $W \leftarrow \widetilde{\mathsf{E}}_{L,4}(X)$
    $W_{\oplus} \leftarrow W_{\ominus} \oplus W$
    $Y \leftarrow \widetilde{\mathsf{E}}_{L,4}(W)$
    $C_j \leftarrow Y \oplus \Delta_N$
  $L \leftarrow L \odot \alpha$
  $X \leftarrow \langle |M_{m-1}| \rangle_n \oplus \Delta_N$
  $W \leftarrow \widetilde{\mathsf{E}}_{L,5}(X)$
  $Y \leftarrow \widetilde{\mathsf{E}}_{L,5}(W)$
  $C_{m-1} \leftarrow \mathsf{chop}(Y \oplus \Delta_N, |M_{m-1}|) \oplus M_{m-1}$
  $W_{\oplus} \leftarrow W_{\oplus} \oplus W \oplus M_{m-1}$
  $C \leftarrow (C_{m-1}, \ldots, C_0)$
  return $(L, W_{\oplus}, C)$

function LOCUS-AEAD-$\widetilde{\mathsf{E}}$.dec(K, N, A, C, T)
  $M \leftarrow \perp$, $W_{\oplus} \leftarrow 0$, $V_{\oplus} \leftarrow 0$
  $(K_N, \Delta_N) \leftarrow \mathsf{init}(K, N)$
  if $|A| \neq 0$ **then**
    $(K_N, V_{\oplus}) \leftarrow \mathsf{proc\_ad}(K_N, \Delta_N, A)$
  if $|M| \neq 0$ **then**
    $(K_N, W_{\oplus}, M) \leftarrow \mathsf{proc\_ct}(K_N, \Delta_N, C)$
  $T' \leftarrow \mathsf{proc\_tg}(K_N, \Delta_N, V_{\oplus}, W_{\oplus})$
  if $T' = T$ **then**
    return M
  else
    return $\perp$

function $\mathsf{proc\_ct}(K_N, \Delta_N, A, C, T)$
  $L \leftarrow K_N$
  $(C_{m-1}, \ldots, C_0) \xleftarrow{n} C$
  for $j = 0$ **to** $m - 2$ **do**
    $Y \leftarrow C_j \oplus \Delta_N$
    $L \leftarrow L \odot \alpha$
    $W \leftarrow \widetilde{\mathsf{E}}^{-1}_{L,4}(Y)$
    $W_{\oplus} \leftarrow W_{\oplus} \oplus W$
    $X \leftarrow \widetilde{\mathsf{E}}^{-1}_{L,4}(W)$
    $M_j \leftarrow X \oplus \Delta_N$
  $L \leftarrow L \odot \alpha$
  $X \leftarrow \langle |C_{m-1}| \rangle_n \oplus \Delta_N$
  $W \leftarrow \widetilde{\mathsf{E}}_{L,5}(X)$
  $Y \leftarrow \widetilde{\mathsf{E}}_{L,5}(W)$
  $M_{m-1} \leftarrow \mathsf{chop}(Y \oplus \Delta_N, |C_{m-1}|) \oplus C_{m-1}$
  $W_{\oplus} \leftarrow W_{\oplus} \oplus W \oplus M_{m-1}$
  $M \leftarrow (M_{m-1}, \ldots, M_0)$
  return $(L, W_{\oplus}, M)$

Slika 10. Prikaz funkcija kriptiranja i dekriptiranja

Dekriptiranje:
Prima ključ K $\in\{0,1\}^{\kappa}$, nonce N $\in\{0,1\}^{\kappa}$,pridružene podatke A $\in\{0,1\}^{*}$, cipher tekst C $\in\{0,1\}^{*}$, tag T $\in\{0,1\}^{n}$ kao ulaz, a vraća plaintext M $\in\{0,1\}^{|C|}$.

Sigurnost:

Guessing the Master Key : $T \approx 2^{\kappa}$
Guessing the Nonce-based Key and Mask : $DT \approx 2^{n+\kappa}$
Online-Online Block Matching : $D^{2} \approx 2^{n+\kappa}$
Online-Offline Block Matching : $DT \approx 2^{n+\kappa}$

2.6 MIXFEED

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	128 bita
metoda nadopune zadnjeg bloka	4 kontrolna bita (t3 ||t2 || t1|| t0)
implementiran algoritam HASH	NE

Namjena i glavna ideja: *Authenticated Encryption with Associated Data* algoritam (u daljnjem tekstu naznačen s AEAD) baziran na AES'128/128 blokovskom kriptografskom algoritmu. Minimally Xored Feedback mode (mixFeed) koristi blokovski kriptografski algoritam u kombinaciji s *key-scheduling* algoritmom. Još zahtijeva i n-bitni xor kako bi obradio n-bitne blokove. Koristi kombinaciju *Plaintext-a* i *Ciphertext-a.* Još jedan od svojstava je da upotrebljava *nonce-dependent key* što povećava razinu sigurnosti u odnosu na konvencionalne modele (poput načina rada otpornog na "curenje" *nonce-dependent key*).

Osnovna svojstva algoritma: mixFeed koristi blokovski kriptografski algoritam parametriziran veličinom bloka n i veličinom ključa κ. Unutar operacije upotrebljava se AES'128/128 (jedina razlika između originalnog AES128/128 je što AES'128/128 koristi u zadnjem koraku *mixcolumn operation*).

Opis algoritma:

1) Algoritam mixFeed

Parametri: ključ duljine 128 bitova, broj koji se koristi samo jednom(*nonce*) duljine 120 bitova, veličina bloka 128 bita

Kada dođe do obrađivanja zadnjeg bloka podataka ulaz blokovskog kriptografskog algoritma ovisi o 4 kontrolna
bita.

Algorithm 2 AES'128/128 Block Cipher. To apply a chain of block cipher, we perform an extra round of AES'128/128 Key-Schedule and use that round key as the initial key of the next call of AES'128/128. As described in the Introduction the second output of Emodule only depends on the first input K and we define this function as $\phi(K)$.

function E$(K; X)$
  $(W_{47}, \ldots, W_0) \leftarrow$ KeyGen(K)
  for $i = 1$ **to** 10 **do**
    $X \leftarrow X \oplus (W_{4i-1}, W_{4i-2}, W_{4i-3}, W_{4i-4})$
    $X \leftarrow$ SubBytes(X)
    $X \leftarrow$ ShiftRows(X)
    $X \leftarrow$ MixColumns(X)
  $X \leftarrow X \oplus (W_{43}, W_{42}, W_{41}, W_{40})$
  $K \leftarrow (W_{47}, W_{46}, W_{45}, W_{44})$
  return (X, K)

function KeyGen(K)
  $(K_{15}, \ldots, K_0) \xleftarrow{8} K$
  for $i = 0$ **to** 3 **do**
    $W_i \leftarrow (K_{4i+3}, K_{4i+2}, K_{4i+1}, K_{4i})$
  for $i = 4$ **to** 47 **do**
    $Y \leftarrow W_{i-1}$
    if $i\%4 = 0$ **then**
      $Y \leftarrow$ SubWords$(Y \lll 8)$
      $Y \leftarrow Y \oplus \mathsf{RCON}_{i/4}$
    $W_i \leftarrow W_{i-4} \oplus Y$
  return $(W_{47}, \ldots, W_0)$

function SubBytes(X)
  $(X_{15}, \ldots, X_0) \xleftarrow{8} X$
  for $i = 0$ **to** 15 **do**
    $X_i \leftarrow \mathrm{AS}(X_i)$
  return X

function Shiftrows(X)
  $(X_{15}, \ldots, X_0) \xleftarrow{8} X$
  for $i = 0$ **to** 3 **do**
    for $j = 0$ **to** 3 **do**
      $Y_{4i+j} \leftarrow X_{4i+((j+i)\%4)}$
  return Y

function MixColumns(X)
  $M \leftarrow \begin{pmatrix} 2 & 3 & 1 & 1 \\ 3 & 1 & 1 & 2 \\ 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix}$
  $Y \leftarrow M \cdot X$
  return Y

Slika 11. Funkcije kriptiranja i dekriptiranja algoritma Mixfeed

Sigurnost:

Sigurnost je jednaka AES 128/128.Otpornost na single-key napade je gotovo jednaka otpornosti na single-key napade.Otpornost na single-key napade poput Bogdanov et al. – 2126

2.7 ACE

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	64 bita
metoda nadopune zadnjeg bloka	10*
Implementiran algoritam HASH	DA

Namjena i glavna ideja: *Authenticated Encryption with Associated Data* algoritam (u daljnjem tekstu naznačen s AEAD) i algoritam HASH. Primarni algoritmi *ACE* familije algoritama su *ACE-AE-128* i *ACE-H-256.* AEAD i HASH koriste isto sklopovlje što ACE čini jednostavnijim za implementaciju u sustavima koji moraju koristiti oba algoritma.

Osnovna svojstva algoritma: *ACE* permutacija s 320-bitnim stanjem. Unutar operacije koristi se *Simeck* kutija. Radi se o permutaciji koja koristi blokovski kripto algoritam *Simeck-64* u 8 iteracija zajedno s dodavanjem konstante. Od operacija po bitovima koristi se isključivo XOR, AND te lijevi kružni pomak i „shuffle“ 64-bitne riječi.

Opis algoritma:

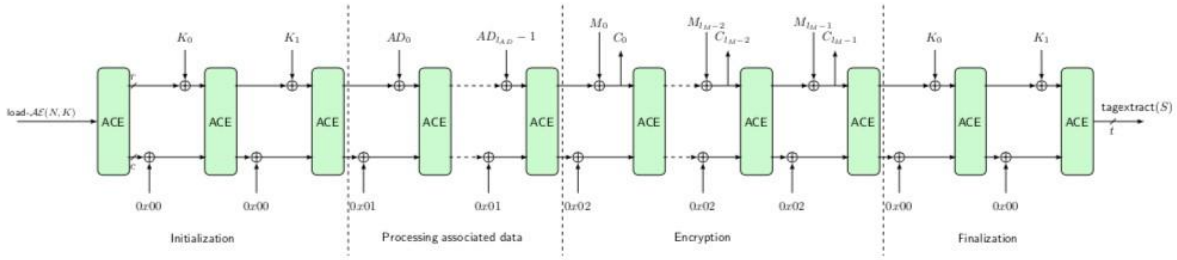
1) ***ACE-AE-128***

Parametri: ključ duljine 128 bitova, *nonce* duljine 128 bitova

I u AEAD i u HASH algoritmu koristi se modificirana spužvasta funkcija. Ona permutaciju s fiksnom duljinom ulaza i izlaza omata u funkciju koja prima ulaz proizvoljne konačne duljine i daje izlaz proizvoljne konačne duljine. Za spužvastu funkciju bitna je širina *bitrate*-a unutar stanja koja se označava s r jer se s prvih *r* bitova obavljaju XOR operacije te se oni prosljeđuju onoj funkciji koja se „omotava“ spužvastom funkcijom. Spužva se sastoji od dvije faze - faze upijanja i faze cijeđenja.

Koraci *ACE-E* dijela AEAD algoritma (*encryption*):

- **učitavanje ključa** i *nonce*-a (jednokratne javne riječi),
- **inicijalizacija stanja** permutacije pomoću ključa i *nonce*-a
- **procesuiranje *Additional data* -** proširivanje (proširuje se s jednim bitom 1, te n bitova 0 do najbližeg višekratnika od 64, poput algoritma Keccak), dijeljenje na blokove (na slici označeni s AD) te ubacivanje u ACE permutaciju blok po blok
- **kriptiranje poruke** – slično prethodnom koraku (blokovi poruke na slici označeni s M) nakon permutacije jednog bloka „cijedi“ se blok šifriranog teksta (metoda *duplex* spužve, za razliku od klasične spužve, ne čeka konačno „upijanje“ svih blokova, već se u potpunosti obrađuje jedan po jedan blok). Nakon dobivenih blokova sve se još jednom ubaci u ACE permutaciju (blokovi kriptiranog teksta označeni su s C)
- **finalizacija –** apsorbiraju se 2 bloka ključa i stvara se *tag* (metoda *tagextract(S)*)



Slika 12. prikaz tijeka kriptiranja unutar AEAD algoritma

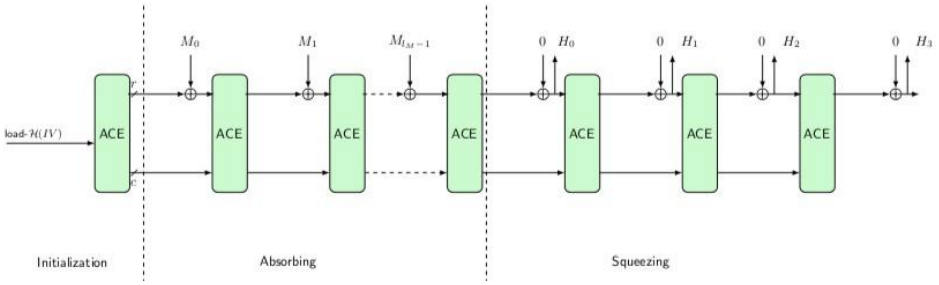
2) ***ACE-H-256***

Parametri: inicijalizacijski vektor duljine 24 bita, sažetak duljine 256 bitova, r (*bitrate*) spužvaste funkcije 64 bita

HASH algoritam koristi spužvastu funkciju prethodno razjašnjenu u opisu AEAD algoritma.

Koraci HASH algoritma:

- **nadopuna posljednjeg bloka** (*padding*) ulazne poruke do prvog sljedećeg višekratnika parametra *r* (proširuje se s jednim bitom 1, te n bitova 0 do najbližeg višekratnika od *r*, poput Keccak algoritma) te dijeljenje poruke na blokove (na slici označene s M_0, M_1 i M_{lm-1})
- **učitavanje** inicijalizacijskog vektora
- **upijanje:** blok po blok poruke se nakon XOR-a i obavljanja ACE permutacije sprema u prvih *r* (64) bitova stanja
- **cijeđenje:** blok po blok izlaza (na slici označeni s H_0 do H_3) dobiva se nakon XOR-a dobivenog stanja i obavljanje ACE permutacije



Slika 13. prikaz tijeka HASH algoritma

Sklopovski zahtjevi: unutar ACE algoritma implementirano je da se sve 3 operacije (enkripcija, dekripcija te hashing) mogu provoditi putem istog sklopovlja. To je proširilo sklopovlje u odnosu na specijalizirano (koje bi obavljalo samo jednu operaciju), no u skladu je sa zahtjevima *lightweight* kriptografije.

Osim toga, za ulaz i izlaz koristi se po jedan *port* zbog smanjenja zahtjeva na sklopovlje.

Dekripcija: simetrična enkripciji, što znači da su koraci dekriptiranja ekvivalentni koracima opisanima u ACE-E algoritmu, samo se korak same dekripcije razlikuje u parametrima – prvih *r* bitova stanja XOR-a se s blokovima kriptirane poruke (označeni s C), a „cijede“ se blokovi originalne poruke.

Sigurnost:

AEAD – otpornost na sve napade: 2^{128}

HASH – otpornost na kolizije: 2^{128}

– otpornost na izračunavanje originala (prva domenska otpornost): 2^{192}

– otpornost na izračunavanje poruke koja daje isti sažetak (druga domenska otpornost): 2^{128}

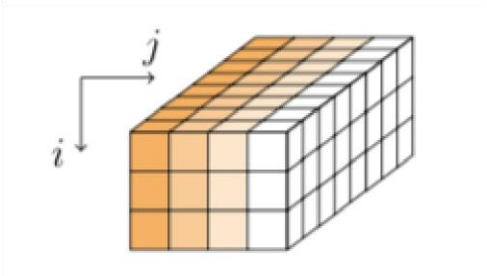
2.8 GIMLI

Specifikacije

veličina ključa primarnog algoritma	256 bitova
alternativne veličine ključeva	-
veličina bloka	128 bitova
metoda nadopune zadnjeg bloka	sljedeći bit XOR 1, zadnji bit XOR 1
implementiran algoritam HASH	DA

Namjena i glavna ideja: *Authenticated Encryption with Associated Data* algoritam (u daljnjem tekstu naznačen s AEAD) i algoritam HASH. Primarni algoritmi iz GIMLI familije su Gimli24v1 (jednak je naziv primarnog AEAD i HASH algoritma). Naziv algoritma povezan je s brojem obavljanja GIMLI permutacije koja je opisana u nastavku. Glavni fokus GIMLI algoritma dobre su performanse u različitim okruženjima. Autori ne garantiraju unaprjeđenje performansi postojećih algoritama na svim platformama, već široko iskoristiv algoritam koji zadovoljava sigurnosne zahtjeve a čija bi univerzalna priroda olakšala standardizaciju lakih kriptografskih algoritama.

Osnovna svojstva algoritma: *GIMLI* permutacija s 384-bitnim stanjem koje se može prikazati pomoću paralepipeda dimenzija 3 x 4 x 32 (Slika 14.) ili alternativno matrice dimenzija 3 x 4 koja pohranjuje 32-bitne riječi. Permutacija, ovisno o broju runde može obavljati jednu od 3 vrste operacija: operacija s SP kutijom, linearno miješanje slojeva (svake druge runde) i dodavanje konstante (svake četvrte runde). Od operacija po bitovima koristi se isključivo AND, OR, XOR te lijevi kružni pomak za konačnu udaljenost i kružni pomaci.



Slika 14. grafički prikaz GIMLI stanja

Opis algoritma:

1) GIMLI AEAD

Parametri: ključ - 256 bitova, *nonce* - 128 bitova, *bitrate* - 128 bitova I u AEAD i u HASH algoritmu koristi se spužvasta funkcija. Ona permutaciju s fiksnom duljinom ulaza i izlaza omata u funkciju koja prima ulaz proizvoljne konačne duljine i daje izlaz proizvoljne konačne duljine. Za spužvastu funkciju bitna je širina *bitrate*-a unutar stanja koja se označava s *r* jer se s prvih *r* bitova obavljaju XOR operacije te se oni prosljeđuju onoj funkciji koja se „omotava" spužvastom funkcijom. Spužva se sastoji od dvije faze - faze upijanja i faze cijeđenja. Za korištenje spužvaste funkcije svi ulazi dijele se na blokove veličine *r*. Za zadnji blok, čija je duljina *b* između 0 i 15 bajtova, operacije su sljedeće:

o blok se XOR-a u prvih *b-1* bajtova GIMLI stanja o XOR 1 u sljedeći bajt stanja (pozicija *b*) o XOR 1 u posljednji bajt stanja (pozicija 47) o obavi se GIMLI permutacija Koraci same enkripcije:

- **učitavanje ključa** i *nonce*-a (jednokratne javne riječi) u GIMLI stanje (*nonce* u prvih 16 bajtova, ključ u sljedećih 32) te izvođenje GIMLI permutacije
- **procesuiranje *Additional data*** - dijeljenje na blokove te ubacivanje u GIMLI permutaciju blok po blok
- **kriptiranje poruke** – slično prethodnom koraku nakon permutacije jednog bloka „cijedi" se blok šifriranog teksta (metoda *duplex* spužve,
  za razliku od klasične spužve, ne čeka konačno „upijanje" svih blokova, već se u potpunosti obrađuje jedan po jedan blok)
- **finalizacija –** nakon što se obradi zadnji (nepotpuni) blok ulaza, *tag* nastaje od prvih 16 bajtova stanja

2) GIMLI HASH

Parametri: sažetak - 256 bitova, *r* (*bitrate*) spužvaste funkcije 128 bitova HASH algoritam koristi spužvastu funkciju prethodno razjašnjenu u opisu AEAD algoritma.

Koraci HASH algoritma:

- **inicijalizacija GIMLI stanja –** stanje se u svim bitovima inicijalizira na 0
- **dijeljenje poruke na blokove** veličine 128 bitova
- **upijanje:** blok po blok poruke se nakon XOR-a i obavljanja GIMLI permutacije sprema u prvih *r* (128) bitova stanja
- **cijeđenje:** prvi blok sažetka dobiva se iz prvih 16 bajtova stanja. Drugi blok dobiva se iz prvih 16 bajtova nakon obavljanja GIMLI permutacije.

Sklopovski zahtjevi: temelj GIMLI enkripcije, dekripcije te algoritma HASH je GIMLI permutacija što uređajima koji bi provodili sve navedene operacije smanjilo vrijeme izvođenja i sklopovske zahtjeve. U samoj permutaciji jednostavne se operacije ponavljaju više puta, što u pogledu sklopovlja omogućava kompaktnost i u smislu *software*-a veću brzinu izvođenja. Za pohranu GIMLI stanja potrebno je 12 registara, a za pohranu privremenih varijabli samo još 2 dodatna registra, što GIMLI čini prikladnim za obavljanje na uređajima ograničene memorije.

Dekripcija: simetrična enkripciji, što znači da su koraci dekriptiranja ekvivalentni koracima opisanima u GIMLI AEAD algoritmu, samo se korak same dekripcije razlikuje u parametrima – prvih *r* bitova stanja XOR-a se s blokovima kriptirane poruke, a „cijede" se blokovi originalne poruke.

Sigurnost:

AEAD – otpornost na sve napade: 2^{128}

HASH – otpornost na sve napade: 2^{128}

2.9 ORANGE

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	n = 128 bitova
metoda nadopune zadnjeg bloka	prefiksiranje s 0*1
Implementiran algoritam HASH	DA

Namjena i glavna ideja: Ime ORANGE dolazi od „*Optimum Rate spoNGE construction*“. Implementirani su AEAD(*Authenticated Encryption with Associated Data*) algoritam ORANGE-Zest te HASH algoritam ORANGEISH. Hash algoritam ORANGEISH je vrlo blizak procesu AD u AE algoritmu ORANGE-Zest kako bi istovremena implementacija oba algoritma bila optimirana.

Osnovna svojstva algoritma: Koristi se PHOTON$_{256}$ permutacija nad stanjem koje se sastoji od 64 elementa po 4 bita svaki, reprezentirano matricom 8x8. PHOTON$_{256}$ se sastoji od 12 rundi, od kojih svaka primjenjuje četiri sloja funkcija (*AddConstant, SubCells, ShiftRows, MixColumnSerial* redom). Funkcija *AddConstant* dodaje fiksnu konstantu ćelijama unutrašnjeg stanja, *SubCells* primjenjuje 4-bitnu S-kutiju nad svakom od 64 4-bitne ćelije, *ShiftRows* rotira pozicije ćelija u svakom redu, a *MixColumnSerial* linearno miješa stupce koristeći serijsko množenje matrica.

```
1: function AddConstant(X, K)
2:     RC[12] ← {1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10}
3:     IC[8] ← {0, 1, 3, 7, 15, 14, 12, 8}
4:     for i = 0 to 7 do
5:         X[i, 0] ← X[i, 0] ⊕ RC[k] ⊕ IC[i]
6:     return X

7: function SubCells(X)
8:     for i = 0 to 7   j=0 to 7 do
9:         X[i, j] ← S-box(X[i, j])
10:    return X

11: function ShiftRows(X)
12:    for i = 0 to 7   j = 0 to 7 do
13:        X′[i, j] ← X[i, (j + i)%4]
14:    return X′
```

```
1: function MixColumnSerial(X)
2:     M ← Serial[2, 4, 2, 11, 2, 8, 5, 6]
3:     M^8 · X
4:     return X

5: function PHOTON_256(X)
6:     for i = 0 to 11 do
7:         X ← AddConstant(X)
8:         X ← SubCells(X)
9:         X ← ShiftRows(X)
10:        X ← MixColumnSerial(X)
11:    return X
```

Slika 15. Pseudokod PHOTON$_{256}$ i funkcija koje koristi

Opis algoritma:

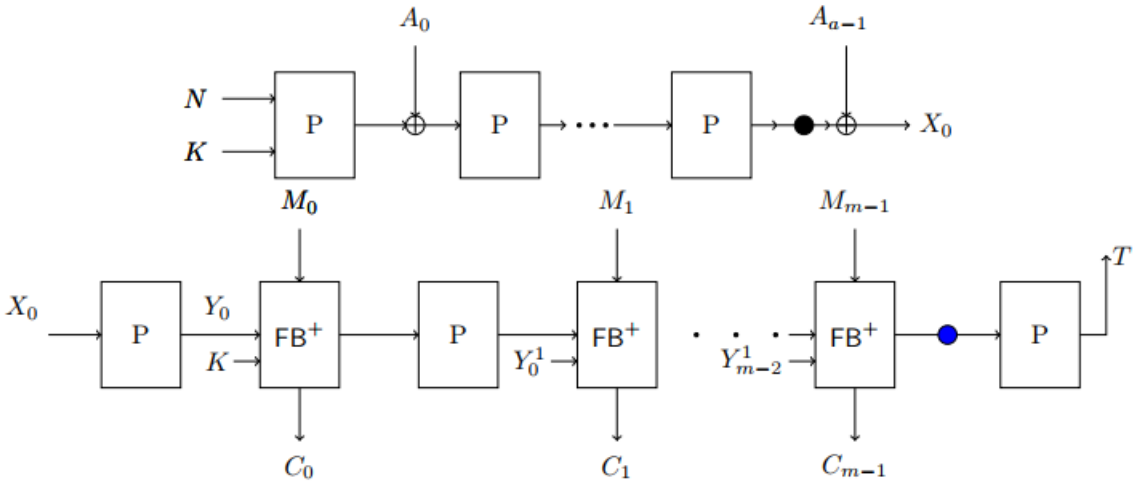
Parametri: ključ duljine 128 bita, *nonce* duljine 128 bita

U algoritmu se koristi modificirana spužvasta funkcija koja može apsorbirati podatke optimalnom brzinom. Spužvasta funkcija permutaciju s fiksnom duljinom ulaza i izlaza omata u funkciju koja prima ulaz proizvoljne konačne duljine i daje izlaz proizvoljne konačne duljine. Za spužvastu funkciju bitna je širina *bitrate*-a (brzina prijenosa) unutar stanja koja se označava s r jer se s prvih r bitova obavljaju XOR operacije te se oni prosljeđuju onoj funkciji koja se „omata“ spužvastom funkcijom. Spužva se sastoji od dvije faze - faze upijanja i faze cijeđenja.

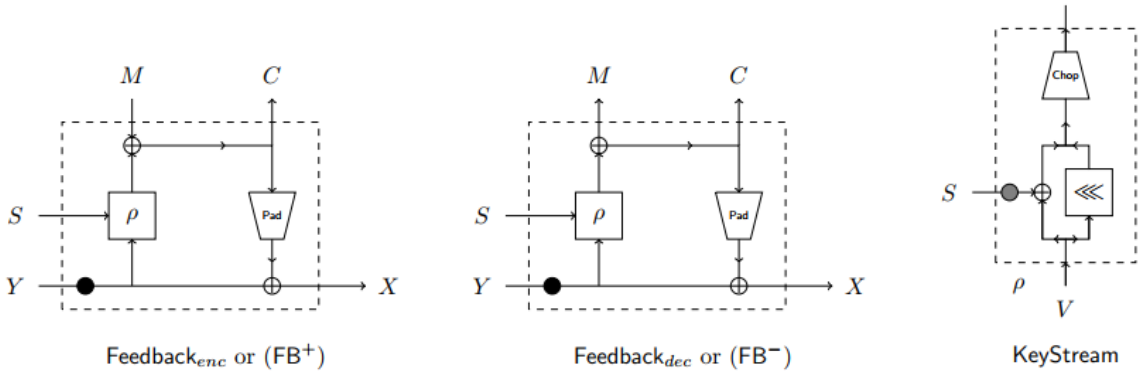
Koraci algoritma:

- Učitavaju se ključ, *nonce*, *AD*, te poruka
- *AD* i poruka razdvajaju se u blokove
- U ovisnosti o duljini poruke i *AD*, inicijaliziraju se stanja permutacije pomoću ključa i *nonce*-a
- Kriptiranje poruke

Algoritam je simetričan, dakle, koraci kriptiranja i dekriptiranja su ekvivalentni. Jedina razlika je u parametrima – funkcija kriptiranja, uz ključ, *nonce* i *AD*, prima M (originalnu poruku) i vraća C (kriptirani tekst), dok je kod funkcije dekriptiranja situacija suprotna



Slika 16. Prikaz rada algoritma u slučaju gdje su duljina poruke i AD različiti od nule



Slika 17. Prikaz rada algoritama FB koji se pozivaju u prikazu na Slici 16

Sigurnost:

Security Model	Data complexity ($\log_2 D$)	Time complexity ($\log_2 T$)
IND-CPA	64	128
INT-CTXT	64	128

Slika 18. Sigurnost algoritma prema navodima u dokumentaciji (D označava podatkovnu složenost napada, a T označava vremensku složenost napada)

2.10 Romulus

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	n = 128 bitova
metoda nadopune zadnjeg bloka X, $\lvert X\rvert \equiv 0 \pmod 8$	$X \parallel 0^{l-\lvert X\rvert-8} \parallel \text{len}_8(X)$
Implementiran algoritam HASH	NE

Namjena i glavna ideja: Romulus je AEAD (*Authenticated Encryption with Associated Data*) baziran na TBC (*Tweakable Block* Cipher) Skinny. Shema Romulus algoritama je podijeljena je u dvije familije:

- Romulus-N → AE baziran na *nonce*-u (NAE), čiji je predstavnik Romulus-N1(koristi Skinny-128-384)
- Romulus-M → AE otporan na zloupotrebu *nonce*-a (MRAE), čiji je predstavnik Romulus-M1 (također koristi Skinny-128-384)

Romulus-N također ima i hardversku implementaciju napisanu u jeziku VHDL.
Primarni cilj Romulusa je pružiti *lightweight*, a istovremeno i AE visoke sigurnosti i učinkovitosti baziran na TBC-u.
Osnovna svojstva algoritma: *beyond-birthday-bound* sigurnost, otpornost na zloupotrebu *nonce*-a, bolje performanse od ΘCB3 algoritma, malen otisak(*footprint*), pogodnost za slanje malih poruka, fleksibilnost

Opis algoritma:

Parametri:

- ključ duljine k = 128 bitova
- *nonce* (*Number used only Once* – broj koji se koristi samo jednom) duljine nl = 128 bitova(kod Romulus-N1 i Romulus-N1) ili 96 bitova (kod ostalih implementacija)
- duljina bloka poruke n = 128 bitova
- duljina brojača d = 56 bitova
- duljina AD bloka $n + t$, gdje je $t \in \{96, 128\}$ (t = 128 kod Romulus-N1 i Romulus-M1)
- oznaka (*tag*) duljine $\tau = 128$ bitova
- TBC, $\tilde{E} : K \times T \times M \rightarrow M$, gdje je $T = T \times B \times D$, $K = \{0, 1\}^k$, $M = \{0, 1\}^n$ $T = \{0, 1\}^t$, $D = [[2^d - 1]]$, i $B = [[256]]_0$. T se koristi za procesuiranje AD ili *nonce*-a, D se koristi za brojač, a B za separaciju domena različitih verzija Romulusa.

Funkcija nadopune posljednjeg bloka:

Za $X \in \{0, 1\}^{s\,l}$ duljine djeljive s 8 (byte string),

$$\boldsymbol{pad_l(X)} = \begin{cases} X, & za\ |\boldsymbol{X}| = \boldsymbol{l} \\ X \,\|0^{l-|X|-8}\, \|\mathrm{len}_8(X), & za\ \boldsymbol{0} < |\boldsymbol{X}| < \boldsymbol{l} \end{cases}$$

LFSR: Kao brojač blokova koristi se LFSR (*Linear-feedback shift register*) te se njegovi rezultati koriste u *Tweaky Encoding* funkciji

Tweaky Encoding: specificiraju se funkcije za TBC, $\tilde{E} : K \times T \times M \rightarrow M$ koristeći Skinny-128-384 (za Romulus-N1 i Romulus-M1) ili Skinny-128-256 (ostali algoritmi). Funkcije su oblika:

$$encode_{m,t}: K \times \mathrm{T} \rightarrow K_T$$

Gdje je $K_T = \{0, 1\}^m$, a m je vrijednost koja ovisi o Skinny verziji (drugi broj u nazivu). Funkcija za Romulus-N1 i Romulus-N2 ($(m, t) = (384, 128)$) glasi:

$$encode_{384,128}(K, T, B, D) = LFSR_{56}(D) \| \mathrm{B} \| 0^{64} \| \mathrm{T} \| K$$

gdje ‖ označava konkatenaciju.

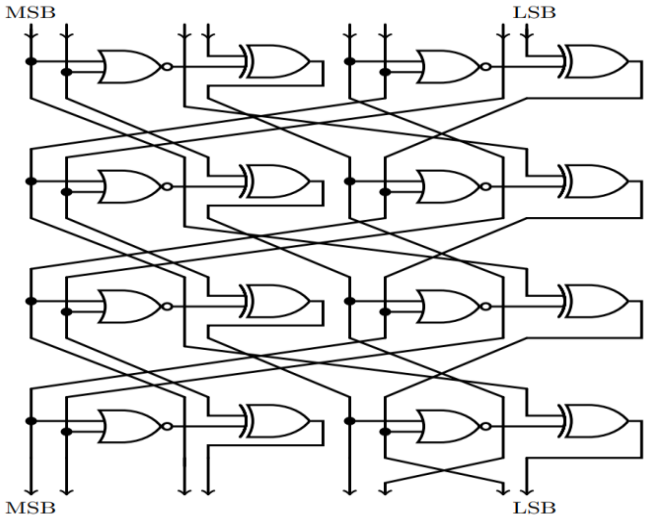
Funkcija ažuriranja stanja temelji se na množenju $n/8 \times n/8$ matrice G koja sadrži 8x8 binarnih podmatrica G_S i niza $X \in \{0, 1\}^n$. Funkcija se definira na način:

$$\rho(S, M) = (S', C)$$

gdje je $C = M \oplus G(S)$ *i* $S' = S \oplus M$. Slično,

$$\rho^{-1}(S, C) = (S', M)$$

gdje je $M = C \oplus G(S)$ *i* $S' = S \oplus M$



Slika 19: sklopovska izvedba S kutije S_8

Algoritam

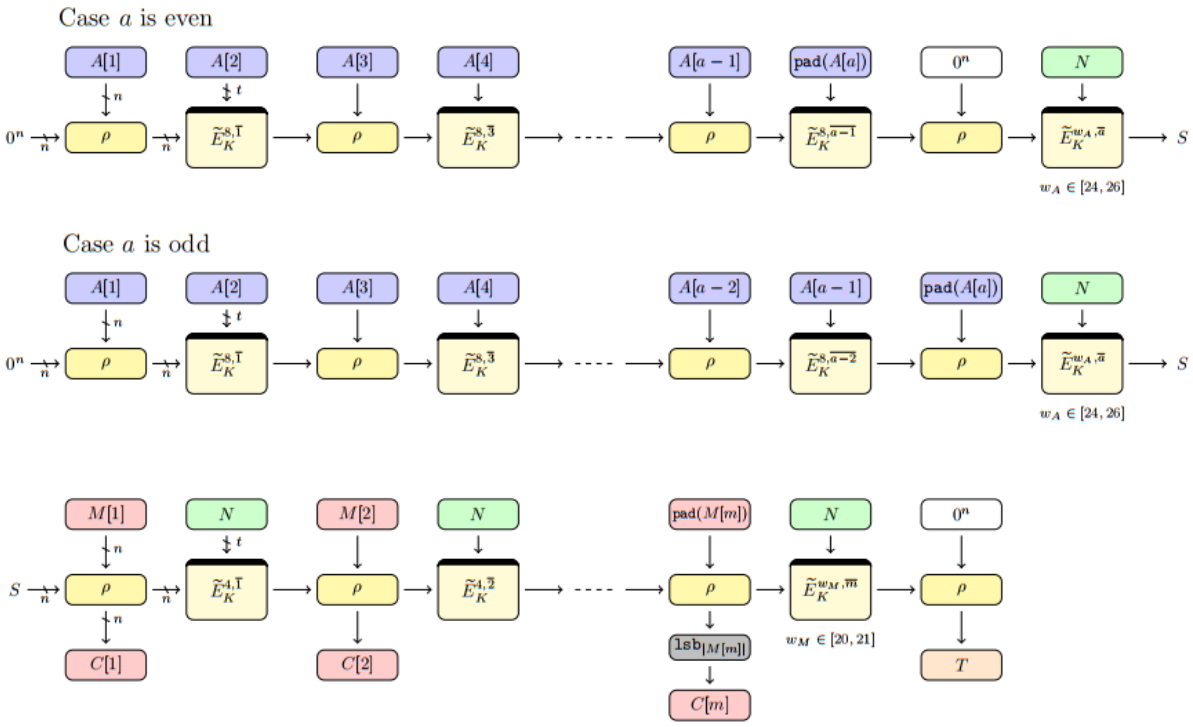
Algorithm Romulus-N.Enc$_K(N, A, M)$
1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** n
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** 24
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
8. $\quad S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
13. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
14. **if** $|M[m]| < n$ **then** $w_M \leftarrow 21$ **else** 20
15. **for** $i = 1$ **to** $m - 1$
16. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
17. $\quad S \leftarrow \widetilde{E}_K^{(N,4,\overline{i})}(S)$
18. **end for**
19. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
20. $(S, C'[m]) \leftarrow \rho(S, M'[m])$
21. $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
22. $S \leftarrow \widetilde{E}_K^{(N,w_M,\overline{m})}(S)$
23. $(\eta, T) \leftarrow \rho(S, 0^n)$
24. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
25. **return** (C, T)

Algorithm Romulus-N.Dec$_K(N, A, C, T)$
1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** n
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** 24
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
8. $\quad S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
13. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
14. **if** $|C[m]| < n$ **then** $w_C \leftarrow 21$ **else** 20
15. **for** $i = 1$ **to** $m - 1$
16. $\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
17. $\quad S \leftarrow \widetilde{E}_K^{(N,4,\overline{i})}(S)$
18. **end for**
19. $\widetilde{S} \leftarrow (0^{|C[m]|} \,\|\, \mathtt{msb}_{n-|C[m]|}(G(S)))$
20. $C'[m] \leftarrow \mathtt{pad}_n(C[m]) \oplus \widetilde{S}$
21. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$
22. $M[m] \leftarrow \mathtt{lsb}_{|C[m]|}(M'[m])$
23. $S \leftarrow \widetilde{E}_K^{(N,w_C,\overline{m})}(S)$
24. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
25. $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
26. **if** $T^* = T$ **then return** M **else** $\perp$

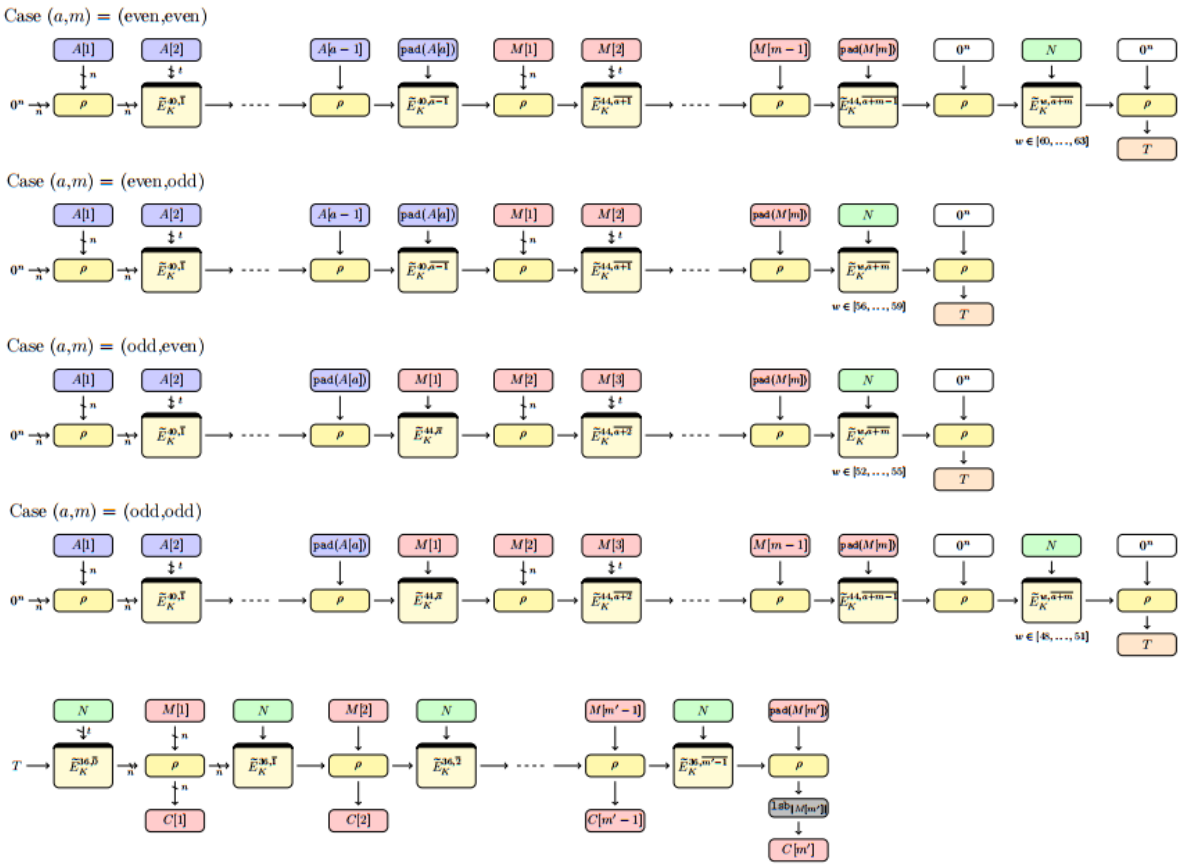
Algorithm $\rho(S, M)$
1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', C)

Algorithm $\rho^{-1}(S, C)$
1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', M)

Slika 20: prikaz pseudokoda algoritama familije Romulus-N



Slika 21: grafički prikaz rada algoritama familije Romulus - N



Slika 22: grafički prikaz rada algoritama familije Romulus-M

Algorithm Romulus-M.Enc$_K(N, A, M)$

1. $S \leftarrow 0^n$
2. $(X[1], \ldots, X[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** n
4. $(X[a+1], \ldots, X[a+m]) \xleftarrow{n+t-u,u} M$
5. **if** $m \bmod 2 = 0$ **then** $v \leftarrow u$ **else** $n+t-u$
6. $w \leftarrow 48$
7. **if** $|X[a]| < u$ **then** $w \leftarrow w \oplus 2$
8. **if** $|X[a+m]| < v$ **then** $w \leftarrow w \oplus 1$
9. **if** $a \bmod 2 = 0$ **then** $w \leftarrow w \oplus 8$
10. **if** $m \bmod 2 = 0$ **then** $w \leftarrow w \oplus 4$
11. $X[a] \leftarrow \mathtt{pad}_u(X[a])$
12. $X[a+m] \leftarrow \mathtt{pad}_v(X[a+m])$
13. $x \leftarrow 40$
14. **for** $i = 1$ **to** $\lfloor (a+m)/2 \rfloor$
15. $\quad (S, \eta) \leftarrow \rho(S, X[2i-1])$
16. $\quad$ **if** $i = \lfloor a/2 \rfloor + 1$ **then** $x \leftarrow x \oplus 4$
17. $\quad S \leftarrow \widetilde{E}_K^{(X[2i], x, \overline{2i-1})}(S)$
18. **end for**
19. **if** $a \bmod 2 = m \bmod 2$ **then**
20. $\quad (S, \eta) \leftarrow \rho(S, 0^n)$
21. **else**
22. $\quad (S, \eta) \leftarrow \rho(S, X[a+m])$
23. $S \leftarrow \widetilde{E}_K^{(N, w, \overline{a+m})}(S)$
24. $(\eta, T) \leftarrow \rho(S, 0^n)$
25. **if** $M = \epsilon$ **then return** (ϵ, T)
26. $S \leftarrow T$
27. $(M[1], \ldots, M[m']) \xleftarrow{n} M$
28. $z \leftarrow |M[m']|$
29. $M[m'] \leftarrow \mathtt{pad}_n(M[m'])$
30. **for** $i = 1$ **to** m'
31. $\quad S \leftarrow \widetilde{E}_K^{(N, 36, \overline{i-1})}(S)$
32. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
33. **end for**
34. $C[m'] \leftarrow \mathtt{lsb}_z(C[m'])$
35. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m'-1] \,\|\, C[m']$
36. **return** (C, T)

Algorithm Romulus-M.Dec$_K(N, A, C, T)$

1. **if** $C = \epsilon$ **then** $M \leftarrow \epsilon$
2. **else**
3. $\quad S \leftarrow T$
4. $\quad (C[1], \ldots, C[m']) \xleftarrow{n} C$
5. $\quad z \leftarrow |C[m']|$
6. $\quad C[m'] \leftarrow \mathtt{pad}_n(C[m'])$
7. $\quad$ **for** $i = 1$ **to** m'
8. $\quad\quad S \leftarrow \widetilde{E}_K^{(N, 36, \overline{i-1})}(S)$
9. $\quad\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
10. $\quad$ **end for**
11. $\quad M[m'] \leftarrow \mathtt{lsb}_z(M[m'])$
12. $\quad M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m'-1] \,\|\, M[m']$
13. $S \leftarrow 0^n$
14. $(X[1], \ldots, X[a]) \xleftarrow{n,t} A$
15. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** n
16. $(X[a+1], \ldots, X[a+m]) \xleftarrow{n+t-u,u} M$
17. **if** $m \bmod 2 = 0$ **then** $v \leftarrow u$ **else** $n+t-u$
18. $w \leftarrow 48$
19. **if** $|X[a]| < u$ **then** $w \leftarrow w \oplus 2$
20. **if** $|X[a+m]| < v$ **then** $w \leftarrow w \oplus 1$
21. **if** $a \bmod 2 = 0$ **then** $w \leftarrow w \oplus 8$
22. **if** $m \bmod 2 = 0$ **then** $w \leftarrow w \oplus 4$
23. $X[a] \leftarrow \mathtt{pad}_u(X[a])$
24. $X[a+m] \leftarrow \mathtt{pad}_v(X[a+m])$
25. $x \leftarrow 40$
26. **for** $i = 1$ **to** $\lfloor (a+m)/2 \rfloor$
27. $\quad (S, \eta) \leftarrow \rho(S, X[2i-1])$
28. $\quad$ **if** $i = \lfloor a/2 \rfloor + 1$ **then** $x \leftarrow x \oplus 4$
29. $\quad S \leftarrow \widetilde{E}_K^{(X[2i], x, \overline{2i-1})}(S)$
30. **end for**
31. **if** $a \bmod 2 = m \bmod 2$ **then**
32. $\quad (S, \eta) \leftarrow \rho(S, 0^n)$
33. **else**
34. $\quad (S, \eta) \leftarrow \rho(S, X[a+m])$
35. $S \leftarrow \widetilde{E}_K^{(N, w, \overline{a+m})}(S)$
36. $(\eta, T) \leftarrow \rho(S, 0^n)$
37. **if** $T^* = T$ **then return** M **else** $\perp$

Algorithm $\rho(S, M)$

1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', C)

Algorithm $\rho^{-1}(S, C)$

1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', M)

Slika 23: prikaz pseudokoda algoritama familije Romulus-M

Sigurnost:

Family	NR-Priv	NR-Auth	NM-Priv	NM-Auth
Romulus-N	128	128	–	–
Romulus-M	128	128	$64 \sim 128$	$64 \sim 128$

Slika 24: sigurnost u ovisnosti o familijama (NR označava *Nonce-Respecting adversary*, NM označava *Nonce-Misusing adversary*)

2.11 Spook

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	256 bita
metoda nadopune zadnjeg bloka	10*
implementiran algoritam HASH	NE

Namjena i glavna ideja: Algoritam je zasnovan na spužvastoj funkciji. Primarno je dizajniran za pružanje sigurnosti protiv napada koji koriste sporedno svojstvo uređaja uz malu potrošnju energije. Kriptiranje je optimizirano za maskirajuće protumjere protiv takvih napada. Spook ima nekoliko istaknutih značajki:

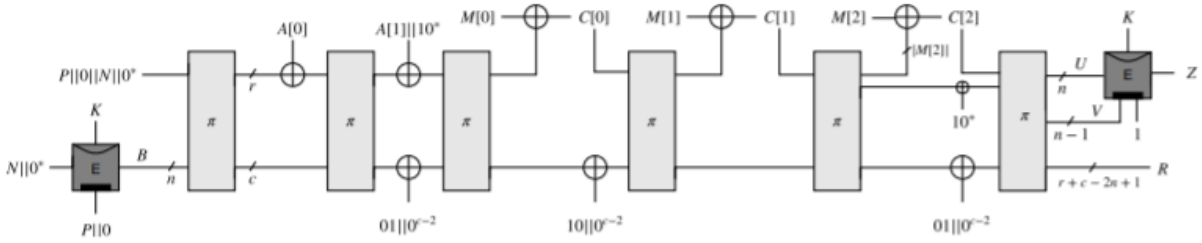
- Otpornost na zloupotrebu *nonce*-a
- Sigurnost protiv rođendanskog napada
- Sigurnost više korisnika uz minimalan trošak

Osnovna svojstva algoritma: Spook se temelji na i specijalizira dva glavna svojstva. Prvi od njih je način rada otporan na „curenje“ koji omogućava učinkovit otpor na napade koji koriste sporedno svojstvo uređaja. Koristi se S1P(Spongent One-Pass) način operacije u tu svrhu. Drugo svojstvo je Clyde-128 TBC (Tweakable Block Cipher) i Shadow-512 permutacija. Oboje od njih su jednostavno proširenje LS-design radnog okvira kojemu je glavni cilj *bitslice* implementacija.

Opis algoritma:

Parametri: ključ duljine 128 bitova, *nonce* duljine 128 bitova

S1P način operacije: Poruka M se parsira u l blokova veličine r. Povezani podaci A se parsiraju u λ blokova na isti način kao i M. Tajni ključ K od n bitova izabire se nasumično u $\{0, 1\}^n$**.** Za kriptiranje, ulazni podaci (poruka, povezani podaci, ključ, *nonce*) se mapiraju iz nizova bajtova u nizove bitova koristeći BMAP funkciju. Za dekriptiranje, zamijene se poruka i kriptirana poruka i koristi se inverzna BMAP funkcija.



Slika 25. S1P način operacije sa TBC permutacijom

Clyde-128 : S1P način operacije zahtijeva TBC. Ovdje se koristi *Tweakable LS-Design (TLS-design)* radni okvir. Rade na n = (s · l) – bitnih stanja, gdje je s veličina od *S-box* i l veličina od *L-box.* X predstavlja potpuno kriptirano stanje. S pogleda implementacije, *S-box* i *L-box* su definirani tako da uvijek mogu biti izvedeni zahvaljujući jednostavnim operacijama nad retcima. TLS osvježava n-bitno stanje x iterirajući N_s koraka. Prednost ovog dizajna je njegova jednostavnost – može biti opisan u nekoliko linija.

Algorithm 1 TLS-design with l-bit L-box and s-bit S-box ($n = s \cdot l$)

$x \leftarrow \mu \oplus TK(0)$; ▷ x is a $s \times l$ bits matrix
for $0 \leq \sigma < N_s$ **do**
  for $0 \leq \rho < 2$ **do**
    $r = 2 \cdot \sigma + \rho$; ▷ Round index
    for $0 \leq j < l$ **do**
      $x[\star, j] = \mathsf{S}(x[\star, j])$; ▷ S-box Layer
    for $0 \leq i < s$ **do**
      $x[i, \star] = \mathsf{L}(x[i, \star])$; ▷ L-box Layer
    $x \leftarrow x \oplus W(r)$; ▷ Constant addition
  $x \leftarrow x \oplus TK(\sigma + 1)$; ▷ Tweakey addition
return x

Slika 26. Opis algoritma

Sigurnost:

Security model	security (bits)
Plaintext confidentiality with nonce misuse-resilience (mR)	$n - \log n$
Ciphertext integrity with misuse-resistance (MR) but no leakage	$n - \log n$
Plaintext confidentiality with encryption leakages and mR	$\approx n/2$
Ciphertext integrity with full leakages and MR	$\approx n - \log n$

Slika 27. Jedno-korisnička sigurnost

2.12 PHOTON-Beetle

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	128 bitova (alternativno 32 bita)
metoda nadopune zadnjeg bloka	0*
implementiran algoritam HASH	DA

Namjena i glavna ideja: PHOTON-Beetle kombinacija je Beetle spužvaste funkcije koja omogućuje implementaciju sa što manje stanja i PHOTON256 permutacije, jednog od najlakših postojećih dizajna. Ta kombinacija dozvoljava jedan od najboljih AEAD dizajna po pitanju broja stanja i površini sklopovlja. PHOTON-Beetle se može klasificirati kao *Authenticated Encryption* familija algoritama, naziva PHOTON-Beetle-AEAD, te kao funkcija sažimanja, naziva PHOTON-Beetle-HASH. Obje familije su parametrizirane *bitrateom* apsorpcije poruke r.

Osnovna svojstva algoritma: *PHOTON* permutacija s 256-bitnim stanjem (*PHOTON*$_{256}$) koja se primjenjuje na stanje od 64 elementa po 4 bita, prikazano 8 x 8 matricom. Permutacija se sastoji od 12 rundi od kojih se svaka sastoji od 4 sloja *AddConstant* (dodavanje konstante internom stanju), *SubCells* (primjena S-kutije na 4-bitne elemente), *ShiftRows*
(rotacija pozicije ćelija u svakom retku) i *MixColumnSerial* (linearno miješanje stupaca matričnim množenjem).

Opis algoritma:

1) **PHOTON-Beetle-AEAD[r]**

Parametri: ključ duljine 128 bitova, *nonce* (*number used only once*, slijed brojeva koji se pridružuju podatcima) duljine 128 bitova, dodatni podatci proizvoljne veličine, poruka proizvoljne veličine, *bitrate r*

Koraci enkriptiranja PHOTON-Beetle-AEAD[r] algoritma:

- **učitavanje ključa** i *nonce-a*
- **inicijalizacija početnog stanja** permutacije povezivanjem ključa na *nonce*
- **procesuiranje dodatnih podataka –** obavlja se identično originalnoj spužvastoj funkciji**,** svakim korakom stanje se ažurira permutacijom i prvih *r* bitova izlaza permutacije (*rate* dio) budu XOR-ani sa sljedećim blokom dodatnih podataka kako bi definirali *rate* dio za sljedeću permutaciju
- **kriptiranje poruke** – Procesuiranje i kriptiranje poruke se odvija na sličan način prethodnom koraku. Kako bi se generirao rezultat enkriptiranja napravi se „*shuffle*“ *rate* dijela izlaza permutacije koji se potom XOR-a s odgovarajućim blokom poruke. Ovaj korak diferencira ovaj postupak od „*Sponge Duplexa*“ (dvostruko sažimanje i cijeđenje) gdje je *rate* dio sljedeće permutacije pušten kao blok kriptiranog teksta. Ažuriranje stanja i generiranje kriptiranog teksta prima funkcija *ρ* koja stvara izlazne podatke koristeći XOR

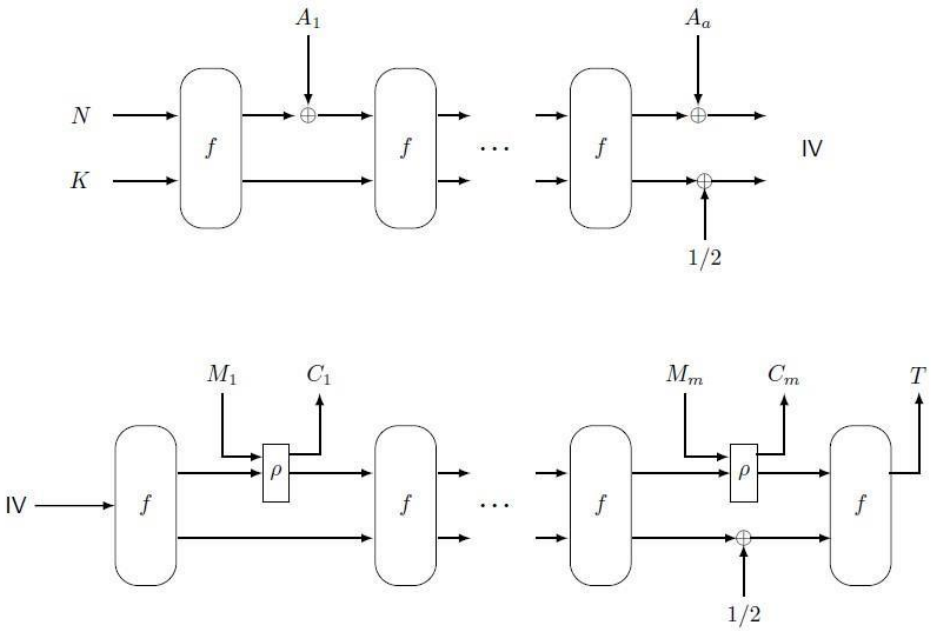
operaciju na izmješanim stanjima i ulaznim podatcima (uz nadopunjavanje ako je blok manji od *r*).

Konačno, dodaju se 3-bitne konstante radi razdvajanja domena. Kod dekriptiranja, funkcija koja prima stanja izlazne podatke kako bi reproducirala ulazne podatke je ρ^{-1}, funkcija inverzna onoj u enkriptiranju.

Preporučene verzije:

a) ***PHOTON-Beetle-AEAD[128] –*** rata apsorpcije je 128, nizak sklopovski otisak s velikom propusnošću

b) ***PHOTON-Beetle-AEAD[32] –*** rata apsorpcije je 32, nizak sklopovski utisak bez davanja dodatne važnosti na propusnosti



Slika 28. PHOTON-Beetle-AEAD.ENC s a AD blokovima i m blokovima poruka

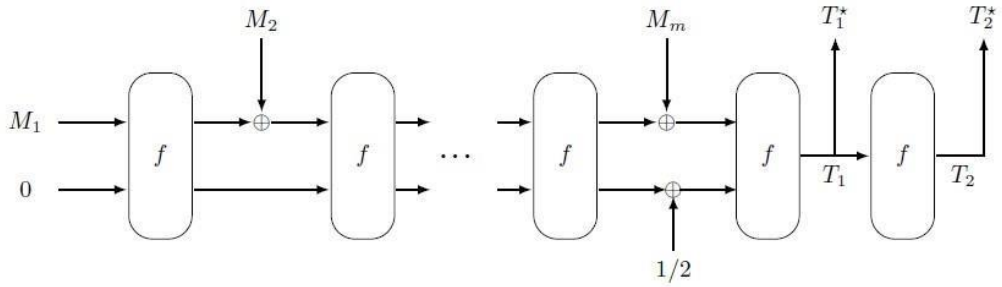
2) PHOTON-Beetle-Hash[r]

- PHOTON-Beetle funkcija sažimanja

Parametri: inicijalizacijski vektor duljine 128 bita

Implementiran ekstremno niskim sklopovskim otiskom s odličnom propusnošću i energetskom učinkovitošću za manje poruke Koraci algoritma funkcije sažimanja:

- **inicijalizacija** stanja pomoću prvih 128 bitova bloka poruke spojenih s nulama radi stvaranja ulaza prve permutacije
- **upijanje:** izlaz svake permutacije se XOR-a sa sljedećim *r*-bitnim blokom poruke s pridodanim nulama kako bi se stvorio ulaz za sljedeću permutaciju. Nakon što zadnji blok poruke bude procesuiran, XOR-a se mala konstanta u dijelu kapaciteta ovisno je li zadnji blok potpun ili djelomičan radi razdvajanja domena
- **cijeđenje:** 256-bitni tag se cijede u 2 dijela od 128 bita
- Preporučena verzija:
  - **PHOTON-Beetle-Hash[32]**



Slika 29. prikaz tijeka algoritma funkcije sažimanja

Sklopovski zahtjevi: Prednost PHOTON-Beetle algoritma je ta da je površina sklopovske implementacije dijelova algoritma jako mala. Korištenje Beetle-a malo povećava cijenu korištene $PHOTON_{256}$ permutacije, jedne od najkompaktnijih *primitiva* tih dimenzija. Sve komponente su izabrane računajući na malu površinu, posebice matrica koja može biti vrlo lako i učinkovito serijalizirana.

S obzirom da se bitova ključa i *nonce-a* ne koriste nakon inicijalizacije, ta memorija se može ponovno iskoristiti. Povećanjem *bitrate-a* ne doalzi do povećanja potrebnih XOR vrata jer su ona serijalizirana.

Sigurnost:

- AEAD – otpornost na napade: 2^{128} za IND-CPA model, 2^{121} za INT-CTXT
- HASH – otpornost na kolizije: 2^{112}

  – otpornost na izračunavanje originala: 2^{128}

2.13 Subterranean 2.0

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	-
veličina bloka	32 bita ako koristi ključ, 8 ako ne koristi
metoda nadopune zadnjeg bloka	10* do 33 bita
implementiran algoritam HASH	DA

Namjena i glavna ideja: Subterranean 2.0 je kriptografski paket za funkciju izračuna sažetka poruke, MAC računanje i autentifikacijsko kriptiranje. Sadrži *duplex* objekt s 257-bitnim stanjem i *lightweight* jednokružnom permutacijom što ga čini dobro prilagođenim za implementacije s malom površinom i niskom energijom u namjenskom sklopovlju.

Osnovna svojstva algoritma: Subterranean 2.0 djeluje na 257-bitnom stanju. Ekstrakcija izlaza *duplex* objekta napravljena je ekstrakcijom 32-bitnog *stringa z* po *duplex* pozivu, gdje je svaki bit od *z* suma 2 bita stanja. Injekcija ulaza obavlja se injekcijom stringa *o* s do 32 bita po *duplex* pozivu u načinu s ključem, a do 8 bita u načinu bez ključa. Središnja funkcija *duplex* objekta je permutiranje stanja, a potom i injekcija ulaznog *stringa* te ekstrakcija izlaza.

Opis algoritma: Subterranean 2.0 definira 3 kriptografske sheme kao načine funkcioniranja temeljene na korištenoj kriptografskoj primitivi, a to su *eXtendable Output Function (XOF)*, *Doubly-Extendable Cryptographic Keyed (deck) function* i *Session Authenticated Encryption (SAE) scheme.* Za NIST-ov natječaj preporučuju *Subterranean-SAE* kao *Authenticated encryption* algoritam i *Subterranean-XOF* kao hash algoritam.

Subterranean 2.0 kružna funkcija R djeluje na 257-bitnom stanju i ima četiri koraka, od kojih svaki korak ima posebnu svrhu: postizanje nelinearnosti, asimetrije, miješanja (*mixing*) i disperzije.

Subterranean *duplex* objekt je jednostavan (interni) *duplex* poziv koji prvo na stanje primjenjuje kružnu funkciju R te potom ubacuje string *o* varijabilne duljine do maksimalno 32 bita. Prije dodavanja u stanje, događa se nadopunjavanje stringa o do 33 bita jednostavnim nadopunjavanjem *(10*)* tako da je rata injekcije (*injection rate*) 33 bita. Između *duplex* poziva moguće je napraviti ekstrakciju 32-bitnog stringa z iz stanja, tako da je rata ekstrakcije (*extraction rate*) 32 bita.
Osim *duplex* poziva i ekstrakcije izlaza, *duplex* objekt ima omotač (*wrapper*) koji se sastoji od tri funkcije koje olakšavaju kompaktnu specifikaciju kriptografskih funkcija i shema. Glavne uloge omotača su podržavanje apsorpcije i cijeđenja *stringova* proizvoljnih duljina i integracija enkriptiranja i dekriptiranja s apsorpcijom. Također nudi separatore između apsorbiranih *stringova* nametanjem zadnjeg injektiranog bloka koji je kraći (ili prazan) od 32 bita u načinu s ključem ili 8 u načinu bez ključa.

Preporučeni algoritmi:

a) **Subterranean-SAE**

Parametri: ključ duljine 128 bitova, *tag* duljine 128 bitova, dodatni podatci i običan tekst limitirani

na *byte stringove*

Algorithm 4 Subterranean-SAE, with τ the tag length

Interface: start(K, N)
$S \leftarrow$ Subterranean()
S.absorb$(K,$ keyed$)$
S.absorb$(N,$ keyed$)$
S.blank(8)

Interface: $(Y, T) \leftarrow$ wrap(A, X, T', op) with op $\in \{\text{encrypt}, \text{decrypt}\}$
S.absorb$(A,$ keyed$)$
$Y \leftarrow S$.absorb$(X,$ op$)$
S.blank(8)
$T \leftarrow S$.squeeze(τ)
if op $=$ decrypt AND $(T' \neq T)$ **then** $(Y, T) = (\epsilon, \epsilon)$
return (Y, T)

Slika 30. Subterranean-SAE algoritam

2) ***Subterranean-XOF***

Subterranean-XOF prima niz proizvoljnog broja *stringova* proizvoljne duljine *M[i]*, prikazanih kao *M[[n]]*, a vraća *bit string* proizvoljne duljine. Koristi se za hashiranje bez ključa.
Ulaz je ograničen na *string* niz jediničnih *byte* stringova proizvoljne veličine, dok je izlaz fiksiran na duljinu 256.

Algorithm 2 Subterranean-XOF

Interface: $Z \leftarrow$ Subterranean-XOF$(M[[n]], \ell)$ with $M[[n]]$ a string sequence and ℓ a natural number
$S \leftarrow$ Subterranean()
for all strings $M[i]$ in $M[[n]]$ **do** S.absorb$(M[i],$ unkeyed$)$
S.blank(8)
return $Z \leftarrow S$.squeeze(ℓ)

Slika 31. Subterranean-XOF algoritam

Sklopovski zahtjevi:

Subterranean 2.0 sklopovski je vrlo učinkovit, no nije prilagođen za programsko ostvarenje, što je u redu s obzirom da je energija po bitu primarni interes na platformama koje su ograničene resursima. S obzirom da se ne rade kompromisi da bi se povećala učinkovitost programskog ostvarenja, Subterranean 2.0 ima iznimno dobar kompromis između sigurnosti i performansa sklopovlja.

Sigurnost:

- Subterranean-XOF – otpornost protiv svih napadača je 2^{112}
- Subterranean-deck – otpornost protiv napadača koji su limitirani na 2^{96} blokova podataka je 2^{128}
- Subterranean-SEA – otpornost protiv svih napadača je 2^{128}

2.14 Xoodyak

Specifikacije

veličina ključa primarnog algoritma	128 bitova
alternativne veličine ključeva	Max. 180 bitova
veličina bloka	128 bitova
metoda nadopune zadnjeg bloka	(00)*
implementiran algoritam HASH	DA

Namjena i glavna ideja: Xoodyak je kriptografski primitiv koji se može koristiti za sažimanje, enkriptiranje , MAC računanje i autentificirano enkriptiranje. Xoodyak podržava dvostruko sažimanje i cijeđenje (engl. *duplex* object) sa sučeljem koji omogućava absorbiranje proizvoljne duljine, enkriptiranje koristeći spužvastu funkciju s fazama upijanja u cijeđenja. Koristi Xoodoo permutaciju, koja svojom širinom od 48 bita omogućuje kompaktne implementacije. Xoodoo je familija permutacija parametrizirana brojem rundi. Stanje Xoodoo permutacije sastoji se od 128 stupaca po 3 bita u 4x32 polju. Svaka permutacija sastoji se od 5 koraka. Snaga sigurnosti je 128 bita

Osnovna svojstva: Xoodyak je svestrani kriptografski objekt pogodan za većinu funkcija sa simetričnim ključevima, uključujući sažimanje, pseudo-random bit generaciju, autentificiranje, enkriptiranje i autentificirano enkriptiranje. Bazira se na *duplex* konstrukciji i na FSKD varijanti kada se koristi sa tajnim ključem. Xoodyak koristi Xoodoo permutaciju. Dizajn se služi 384-bitnom permutacijom inspiriranom s Keccak-p, štoviše dimenzioniran je poput Gimli algoritma zbog djelotvornosti na *low-end* procesorima. Način rada na kojem se bazira Xoodoo naziva se Cyclist.

Opis algoritma: Načini rada koje Xoodyak pruža: način sažimanja i način rada ključa koji se biraju pri inicijalizaciji..

1. Način rada sažimanja: Parametri: string X.

ABSORB(X) absorbira uaz stringa X, dok SQUEEZE(l) stvara l-bajtni izlaz zavisan o dosadašnjim absorbiranim podacima. Prikaz osnovnog slučaja slijedi:

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$ {initialization in hash mode}
$\textsc{Absorb}(x)$ {absorb string x}
$h \leftarrow \textsc{Squeeze}(n)$ {get n bytes of output}

Slika 32. Prikaz hash modea

h dobiva *n*-bajtni *digest* od x, gdje n odabire korisnik.

2. Način rada ključa: **Parametri:** ključ K i poruka M.

U ovome načinu rada Xoodyak radi *stream* enkriptiranje, MAC računanje i autentificirano enkriptiranje. Sljedeća slika prikazuje MAC računanje gdje je ulazni parameter poruka M i ključ K. Izlaz daje t-bajtni tag, najčešće odabrano na 16 bajta (128 bita).

$\textsc{Cyclist}(K, \epsilon, \epsilon)$ {initialization in keyed mode with key K}
$\textsc{Absorb}(M)$ {absorb message M}
$T \leftarrow \textsc{Squeeze}(t)$ {get tag T}

Slika 33. *MAC računanje*

Nadalje enkriptiranje se radi na *stream cipher* način, stoga zahtjeva slučajni broj korišten jednom (engl. *nonce)*. Enkriptiranje poziva SQUEEZE() i koristi izlaz kao *keystream*. ENCRYPT(P) radi na sličan način, ali također absorbira *P* blokova dok se vrši enkriptiranje.

$\textsc{Cyclist}(K, \epsilon, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$C \leftarrow \textsc{Encrypt}(P)$ {get ciphertext C}

Slika 34. Prikaz enkriptiranja

Za dekriptiranje potrebno je sa prikaze Slike 3. zadnju liniju zamijeniti sa

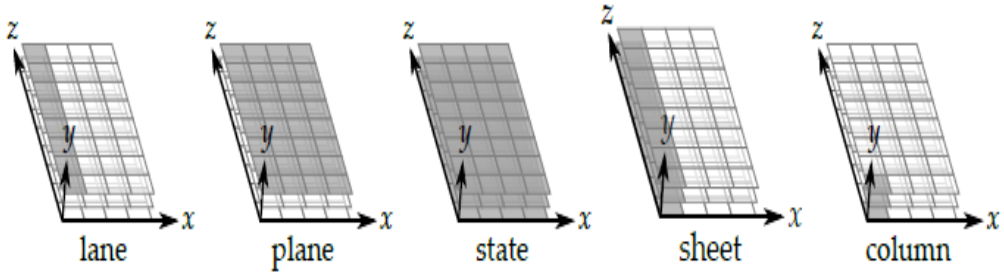
P <- DECRYPT(C) { get plaintext P}

Autentificirano enkriptiranje ostvaruje se kombinacijom prethodne dvije slike

$\textsc{Cyclist}(K, \epsilon, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$\textsc{Absorb}(A)$ {absorb associated data A}
$C \leftarrow \textsc{Encrypt}(P)$
$T \leftarrow \textsc{Squeeze}(t)$ {get tag T}

Slika 35. Prikaz autentificiranog enkriptiranja

Cjeloviti prikaz Xoodoo permutacije na kojoj se zasniva rad Xoodyak algoritma dan je u nastavku:



Slika 36. Prikaz Xoodoo stanja sa stupcima reduciranim na 8 bita i osjenčanim različitim dijelovima stanja

Table 1: Notational conventions

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Algorithm 1 Definition of XOODOO$[n_r]$ with n_r the number of rounds

Parameters: Number of rounds n_r
for Round index i from $1 - n_r$ to 0 **do**
$A = \mathrm{R_i}(A)$

Here $\mathrm{R_i}$ is specified by the following sequence of steps:

θ :
$P \leftarrow A_0 + A_1 + A_2$
$E \leftarrow P \lll (1, 5) + P \lll (1, 14)$
$A_y \leftarrow A_y + E$ for $y \in \{0, 1, 2\}$

ρ_{west} :
$A_1 \leftarrow A_1 \lll (1, 0)$
$A_2 \leftarrow A_2 \lll (0, 11)$

ι :
$A_0 \leftarrow A_0 + \mathrm{C}_i$

χ :
$B_0 \leftarrow \overline{A_1} \cdot A_2$
$B_1 \leftarrow \overline{A_2} \cdot A_0$
$B_2 \leftarrow \overline{A_0} \cdot A_1$
$A_y \leftarrow A_y + B_y$ for $y \in \{0, 1, 2\}$

ρ_{east} :
$A_1 \leftarrow A_1 \lll (0, 1)$
$A_2 \leftarrow A_2 \lll (2, 8)$

Slika 37. Xoodoo implementacija

Sve korištene slike preuzete su s: https://csrc.nist.gov/projects/lightweight-cryptography/round-1-candidates

3. Tehničke značajke

Prije početka izrade programske potpore morali smo se uvjeriti da odabrani algoritmi rade prema očekivanju.

3.1 Prevođenje algoritama

U NIST-ovom opisu natječaja navedeno je da za prevođenje algoritama koriste GCC 5.4.0 s određenim zastavicama:

```
-std=c99 -Wall -Wextra -Wshadow -fsanitize=address,undefined -O2
```

Da bi testirali ispravnost prijavljenih algoritama trebalo je generirati ispitne vektore. U datoteci **genkat_aead.c** nalazi se metoda *generate_test_vectors* koja pokreće *crypto_aead_encrypt* metodu i zatim na istom primjeru pokreće *crypto_aead_decrypt* metodu i uspoređuje njihove rezultate - *decrypt* metoda preko pointera vraća PT (Plain text) koji mora biti jednak PT-u koji je predan *crypt* metodi. Važno je primjetiti da algoritmi moraju raditi za različite duljine plain texta (čak i 0).
Uspješno izvedena metoda završava bez grešaka i rezultira u .txt dokumentu koji ima sljedeću strukturu

```
…
Count = 35
Key = 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Nonce = 000102030405060708090A0B0C0D0E0F
PT = 00
AD = 00
CT = D44BFEBDB1382A0FAC73BC9CDD0657C762
```

… gdje je s PT označen Plain Text, s AD Additional Data i s CT Cypher Text (rezultat kriptiranja).

Struktura mape za algoritam MojAlgoritam morala bi izgledati ovako:

MojAlgoritam
- **Documents**
- **Implementations**
  - **crypto_aead** → **MojAlgoritamV1**
  - **generiranTestniVektor.txt (1*)**
  - **ref**
    - **crypto_aead.h**
    - **genkat_aead.c**
    - **api.h**
    - **encrypt.c**
    - **Makefile (proizvoljno) (2*)**
    - **…**
    - **add_compact (proizvoljno)**
    - **…**
  - **crypto_hash (proizvoljno)**

Glavni algoritam nalazi se u direktoriju *ref*, no moguće je ponuditi više familija algoritma. U valjanoj prijavi se unutar mape glavnog algoritma nalazi .txt datoteka s generiranim ispitnim vektorima (1*) , no i za glavni algoritam i ostale familije algoritama potrebno je generirati ispitne vektore.

S obzirom na to da se kriptiranje može razbiti u više .c datoteka bitno je obuhvatiti ih sve prilikom prevođenja.
Naredbe za prevođenje ovise o (ne)prisutnosti datoteke *Makefile* (označen s 2*).

METODA 1. - PRISUTAN MAKEFILE

Makefile je datoteka bez ekstenzije koja u sebi sadrži dodatne upute za jezični prevoditelj (u ovom slučaju GCC) i olakšava prevođenje više datoteka istovremeno.
Ako postoji u direktoriju moguće je korištenje naredbe ***make*** ___ gdje na praznu crtu idu „naredbe“ koje su navedene u datoteci *Makefile*.

Struktura *Makefile*-a razlikuje se od algoritma do algoritma.

```
.PHONY: clean

CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -Wshadow -fsanitize=address,undefined -O2

all: hash aead

aead: subterranean_ref.c crypto_aead.c genkat_aead.c
  $(CC) $(CFLAGS) -o genkat_aead subterranean_ref.c crypto_aead.c genkat_aead.c

hash: subterranean_ref.c crypto_hash.c genkat_hash.c
  $(CC) $(CFLAGS) -o genkat_hash subterranean_ref.c crypto_hash.c genkat_hash.c

clean:
  @echo "Cleaning up..."
  rm -f "genkat_hash"
  rm -f "LWC_HASH_KAT_256.txt"
  rm -f "genkat_aead"
  rm -f "LWC_AEAD_KAT_128_128.txt"
  @echo "Cleaning done."
```

Slika 38. Primjer Subterennean

U ovom *Makefile*-u definirane su „naredbe“ *all*, *aead*, *hash* i *clean*.
Kako bi generirali testne vektore za ovaj algoritam, pozicionirani u *ref* direktoriju potrebno je pokrenuti naredbu ***make aead*** i zatim pokrenuti program koji se generirao (u ovom slučaju **genkat_aead**) da bi se stvorila .txt datoteka. Datoteku i .exe program moguće je izbrisati naredbom ***make clean***.

```
LWC_AEAD_KAT_256_128.txt: kat
  ./run

kat: Makefile nist/kat_aead.c cipher.c
  ./build
```

Slika 39. Primjer za Gimli

Ovdje su specifirane skraćene naredbe *build* i *run,* što znači da je moguće koristiti prvo ./build, pa ./run.
Moguće je i koristiti naredbu ***make kat*** za prevođenje i ***make LWC_AEAD_KAT_256_128.txt*** za pokretanje programa koji generira testne vektore.

METODA 2. – NIJE PRISUTAN MAKEFILE

Makefile u nekim mapama algoritama nije prisutan što znači da je te algoritme potrebno prevesti postepeno.
Ako se kod za kriptiranje nalazi samo u *encrypt.c* datoteci, prevođenje bi izgledalo ovako:

```
gcc -o genkat genkat_aead.c encrypt.c
```

Nakon ovoga pokreć se **genkat** program koji generira .txt datoteku.
Ako u mapi postoji više .c datoteka sve ih je potrebno navesti kod prevođenja.

3.2 Izrada .dll datoteka

Kako bi se napisani algoritmi u .c i .cpp formatu mogli pokretati iz izrađenog programskog sučelja u Javi potrebno je stvoriti .dll datoteke odabranih algoritama.

Prilikom izrade .dll datoteka koristili smo clang/clang++ prevoditelj, koji je dostupan: http://releases.llvm.org/download.html#9.0.0, Win64 verzija pod „Pre-Built Binaries“.
STVARANJE JAVA HEADERA
Prvo je potrebno stvoriti header datoteku na temelju koje ćemo kasnije stvoriti .cpp datoteku. Java klasa koju je potrebno stvoriti dana je u nastavku

```java
package algoritmi.romulus;

public class RomulusH{

    static{
        System.loadLibrary("nativeRomulus");
    }

    public native byte[] encrypt(byte[] message, long messageLength,
                                 byte[] aData, long aDataLength,
                                 byte[] nonce, byte[] key);

    public native byte[] decrypt(byte[] cipher, long cipherLength,
                                 byte[] aData, long aDataLength,
                                 byte[] nonce, byte[] key);
}
```

Ključna riječ `native` u deklaraciji metoda označava da će one biti definirane u „native“ shared lib čiji je izvorni kod, u našem slučaju napisan u C-u. byte[] koristimo iz razloga što je lakše pretvarati iz `byte[]` u `char*` nego iz `String` u `char*`.
Tu klasu potrebno je prevesti naredbom:
`javac -h . imeKlase.java`

Time će se stvoriti datoteka sa zaglavljem na temelju koje je potrebno opisati navedene metode u novoj .cpp datoteci. To bi zaglavlje trebalo izgledati ovako:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class RomulusH */

#ifndef _Included_RomulusH
#define _Included_RomulusH
#ifdef __cplusplus
extern "C" {
#endif
/*
* Class:     RomulusH
* Method:    encrypt
* Signature: ([BJ[BJ[B[B)[B
 */
JNIEXPORT jbyteArray JNICALL Java_RomulusH_encrypt
  (JNIEnv *, jobject, jbyteArray, jlong, jbyteArray, jlong, jbyteArray, jbyteArray);

/*
* Class:     RomulusH
* Method:    decrypt
* Signature: ([BJ[BJ[B[B)[B
 */
JNIEXPORT jbyteArray JNICALL Java_RomulusH_decrypt
  (JNIEnv *, jobject, jbyteArray, jlong, jbyteArray, jlong, jbyteArray, jbyteArray);

#ifdef __cplusplus
}
#endif
#endif
```

Definicija metoda

Prvo je potrebno stvoriti novu .cpp datoteku čije ime odgovara imenu zaglavlja, to jest datoteke .h.

```
#include "algoritmi_romulus_RomulusH.h"
#include "crypto_aead.h"
#include <malloc.h>

#define CRYPTO_ABYTES 16            //iz api.h
#define MAX_MESSAGE_LENGTH 32       //iz genkat_aead.c


extern "C"{
    JNIEXPORT jbyteArray JNICALL Java_algoritmi_romulus_RomulusH_encrypt
    (JNIEnv* env, jobject thisObject, jbyteArray message,
    jlong mLength, jbyteArray aData, jlong aDataLength,
    jbyteArray nonce, jbyteArray key){

        jboolean isCopy;

        unsigned char* m = (unsigned char*) env->GetByteArrayElements(message, &isCopy);
        unsigned long long mlen = (unsigned long long) mLength;

        unsigned char* ad = (unsigned char*) env->GetByteArrayElements(aData, &isCopy);
        unsigned long long adlen = (unsigned long long) aDataLength;

        unsigned char* npub = (unsigned char*) env->GetByteArrayElements(nonce, &isCopy);

        unsigned char* k = (unsigned char*) env->GetByteArrayElements(key, &isCopy);

        unsigned char* c;
        c = (unsigned char*) malloc ((MAX_MESSAGE_LENGTH + CRYPTO_ABYTES) * sizeof (char));
        unsigned long long clen = 0;

        int unsuccessful = crypto_aead_encrypt(c, &clen, m, mlen, ad, adlen, NULL, npub, k);

        jbyteArray crypted = (*env).NewByteArray(clen);
        (*env).SetByteArrayRegion(crypted, 0, clen, (jbyte*) c);
        free(c);

        return crypted;
    }

    JNIEXPORT jbyteArray JNICALL Java_algoritmi_romulus_RomulusH_decrypt
    (JNIEnv* env, jobject thisObject, jbyteArray crypted,
    jlong cLength, jbyteArray aData, jlong aDataLength,
    jbyteArray nonce, jbyteArray key){
```

```
        jboolean isCopy;

        unsigned char* c = (unsigned char*) env->GetByteArrayElements(crypted, &isCopy);
        unsigned long long clen = (unsigned long long) cLength;

        unsigned char* ad = (unsigned char*) env->GetByteArrayElements(aData, &isCopy);
        unsigned long long adlen = (unsigned long long) aDataLength;

        unsigned char* npub = (unsigned char*) env->GetByteArrayElements(nonce, &isCopy);

        unsigned char* k = (unsigned char*) env->GetByteArrayElements(key, &isCopy);

        unsigned char* m;
        m = (unsigned char*) malloc ((MAX_MESSAGE_LENGTH + CRYPTO_ABYTES) * sizeof (char));
        unsigned long long mlen = 0;

        int unsuccessful = crypto_aead_decrypt(m, &mlen, NULL, c, clen, ad, adlen, npub, k);

        jbyteArray decrypted = (*env).NewByteArray(mlen);
        (*env).SetByteArrayRegion(decrypted, 0, mlen, (jbyte*) m);
        free(m);

        return decrypted;
    }
}
```

Budući da obje metode/funkcije funkcioniraju gotovo identično, dovoljno je objasniti samo jednu. Pošto se tipovi podataka u Javi te tipovi podataka u C/C++ međusobno razlikuju što možemo i vidjeti iz ulaznih argumenata. Te argumente je stoga potrebno svesti na oblik prihvatljiv C/C++-u.
Prvo je potrebno pretvoriti `jbyteArray` (što u Javi odgovara `byte[]`) u `unsigned char*` oblik, a `jlong` (u Javi samo `long`) u `unsigned long long`, budući da to zahtijevaju prototipi funkcija navedeni u `crypto_aead.h`.

Pomoću metode `GetByteArrayElements` pretvaramo `jbyteArray` u `unsigned char*`. Cast je nužan budući da je povratna vrijednost te metode `char*`. Ona kao argumente prima odgovarajući `jbyteArray` te pointer na `jboolean` varijablu u koju se sprema uspješnost pretvaranja pa je stoga potrebno isti definirati prije prvog poziva metode.
Tip varijable `jlong` može se jednostavno samo castati u `unsigned long long`.
Budući da funkcije enkripcije i dekripcije ne vraćaju sam tekst kao povratnu vrijednost, već `int` u koji će biti jednak nuli ukoliko je enkriptiranje/dekriptiranje uspješno obavljeno, kao argument moramo slati još jedan dodatni `unsigned char*` i `unsigned long long*` u koje će se spremati kriptirani tekst te njegova duljina. Stoga je potrebno alocirati varijable u koje će se te vrijednosti spremati. Nakon toga možemo pozvati funkciju za enkripciju/dekripciju s odgovarajućim argumentima, pri čemu je vrijednost argumenta nsec iz prototipa obiju funkcija jednaka `NULL`.
Nakon što je izvršena funkcija te je tekst enkriptiran/dekriptiran, potrebno ga je pretvoriti u Javi prihvatljiv oblik, odnosno `jbyteArray`. Prvo je potrebno pomoću metode `NewByteArray` stvoriti `jbyteArray`

potrebne veličine te potom pomoću metode `SetByteArrayRegion` prekopirati vrijednosti (parametri su redom: `jbyteArray` u koji želimo kopirati, početni indeks, konačni indeks te `unsigned char*` iz kojeg kopiramo).

Nakon što je napravljen most između Jave i C-ovih funkcija kriptiranja, potrebno je sve .cpp i .c fileove zapakirati u .dll file koji će Java znati čitati sljedećimm naredbama:

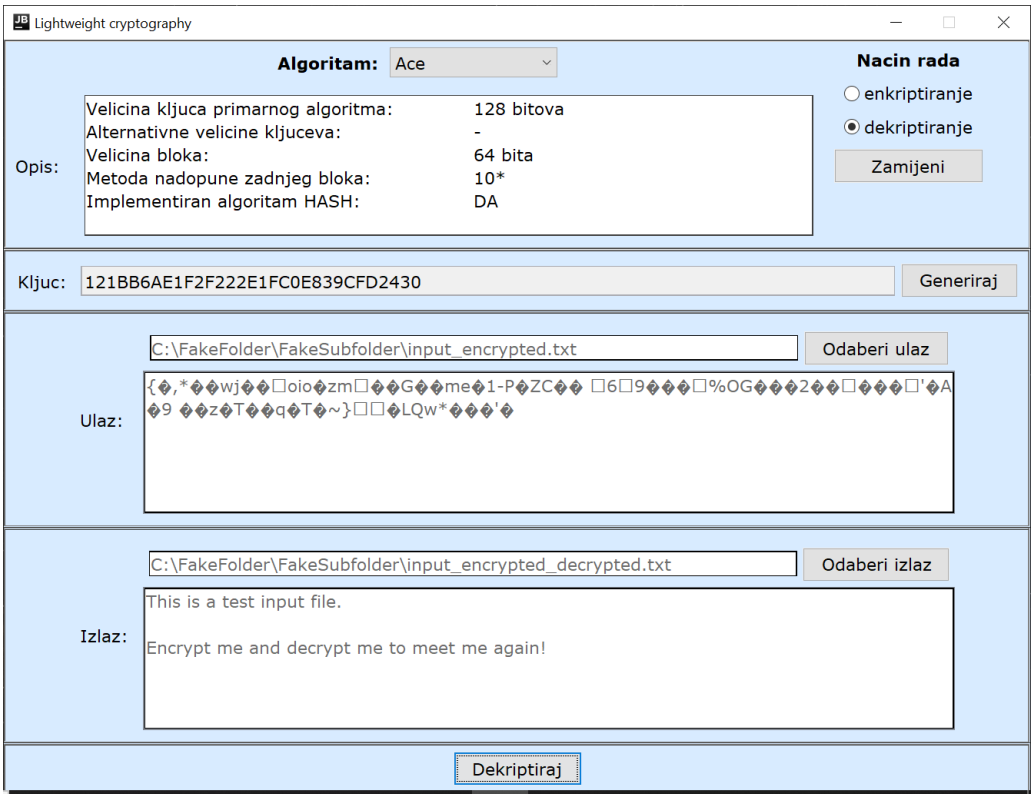
```
clang++ -c "-I%JAVA_HOME%\include" "-I%JAVA_HOME%\include\win32" imeKlase.cpp
-o imeKlase.o
```

Nakon što su tako prevedeni svi `.c/.cpp` fileovi, potrebno ih je zamotati u `.dll` oblik naredbom:

```
clang++ -shared -o native.dll ime1.o ime2.o –Wl
```

gdje na mjesto `ime1.o ime2.o` treba navesti sve .o datoteke.
Time je stvoren .dll datoteka, ali je još potrebno napisati Java kod kojim se pozivaju metode iz klase , primjerice `RomulusH.java` te se ulazni podaci pretvaraju u traženi oblik (`byte[]`).

4. Upute za korištenje



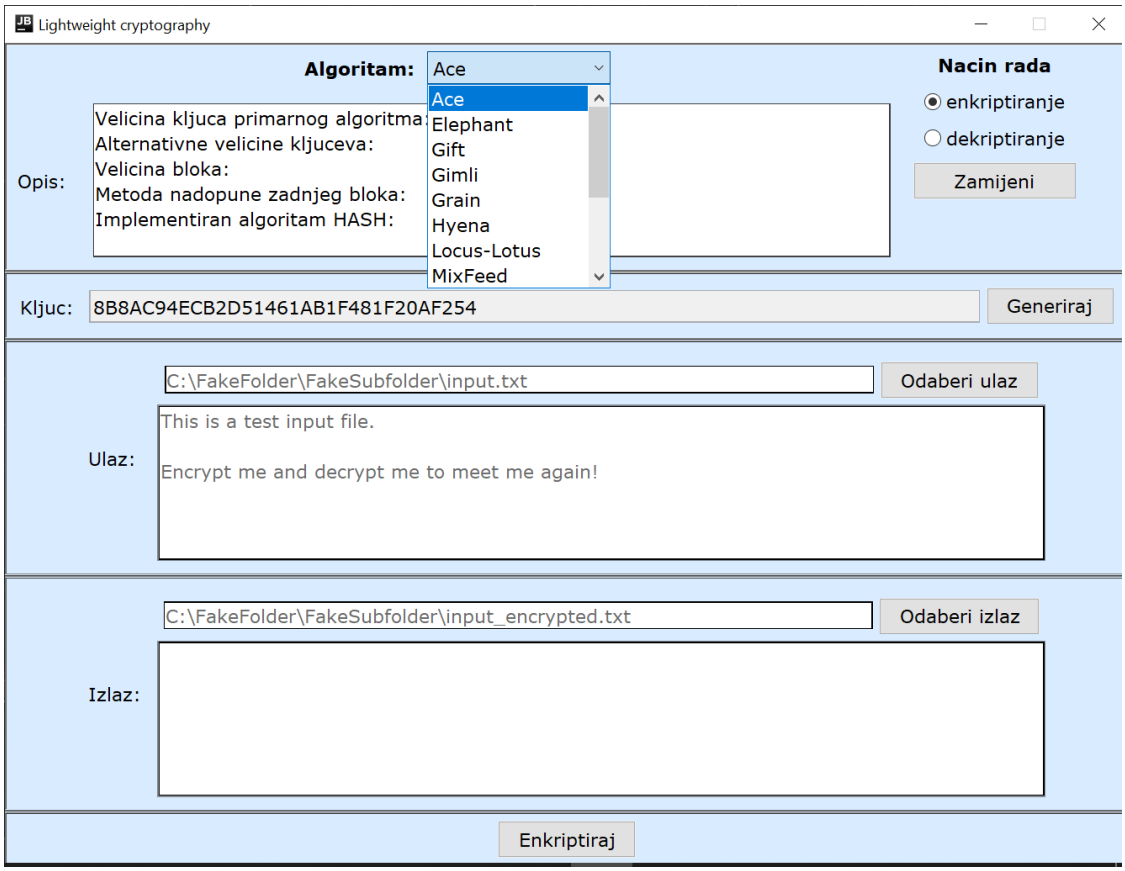
Slika 40. Grafičko sučelje

Grafičko sučelje ima oblik jednog prozora koji je podijeljen na pet dijelova. Prvi prozor sastoji se od padajuće liste algoritama, tekstualnog prozora za opis odabranog algoritma, gumbova za odabir načina rada i gumb za zamjenu ulazne datoteke izlaznom. Drugi dio sadrži tekstualni prozor za prikazivanje ključa i gumb za generiranje istog. Treći i četvrti prozor imaju jednake komponente, a to su tekstualni prozor za prikaz adrese i imena datoteke ulaza ili izlaza, gumb za odabir datoteke te prozor za prikaz sadržaja iste datoteke. Treći prozor služi za odabir i prikaz sadržaja ulazne datoteke, dok četvrti za odabir izlazne datoteke i prikaz rezultata kriptiranja. Peti dio sadrži samo gumb za pokretanje postupka kriptiranja – enkriptiranja ili dekriptiranja, ovisno o odabranom načinu rada. Svi tekstualni dijelovi kodiranu su prema US-ASCII formatu.

Programsko sučelje može se koristiti za kriptiranje i dekriptiranje, no način korištenja se ne razlikuje po funkciji koja se provodi.
Za korištenje je potrebno:

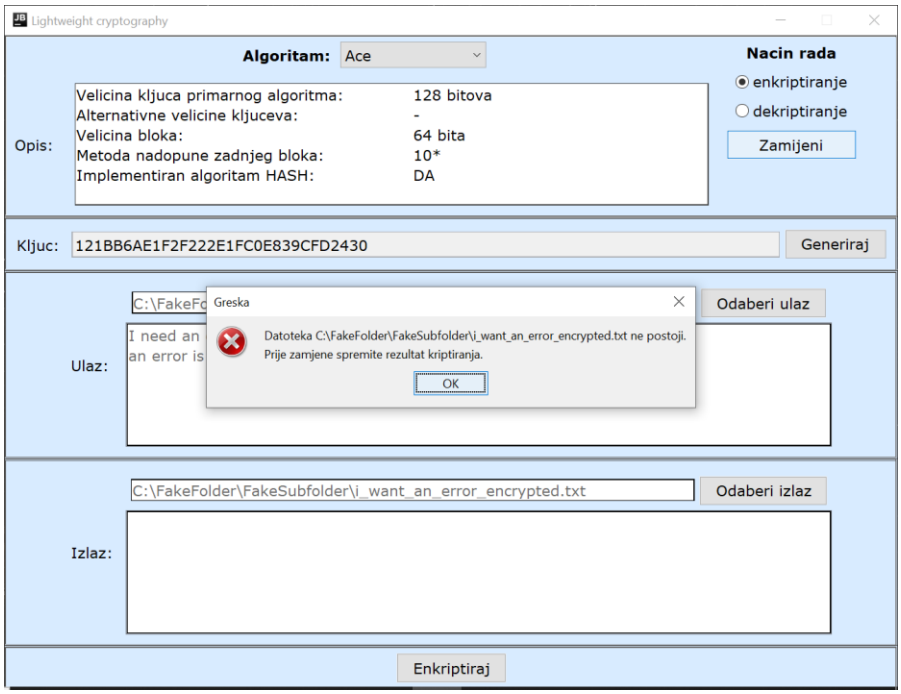
1. Odabrati željeni algoritam
2. Odabrati način rada – enkriptiranje ili dekriptiranje
3. Slučajno generirati ključ
4. Odabrati ulaznu datoteku
5. Odabrati izlaznu datoteku ako se ne želi koristiti predodređena datoteka u koju će se spremati izlaz
6. Stisnuti gumb za provođenje kriptiranja

Pri pokretanju programa, svi parametri su već postavljeni na predodređene vrijednosti, uz iznimku ključa koji je postavljen na nasumično generiranu vrijednost. Kriptiranje se može provesti bez ikakvih promjena ili uz promjene samo nekih parametara.



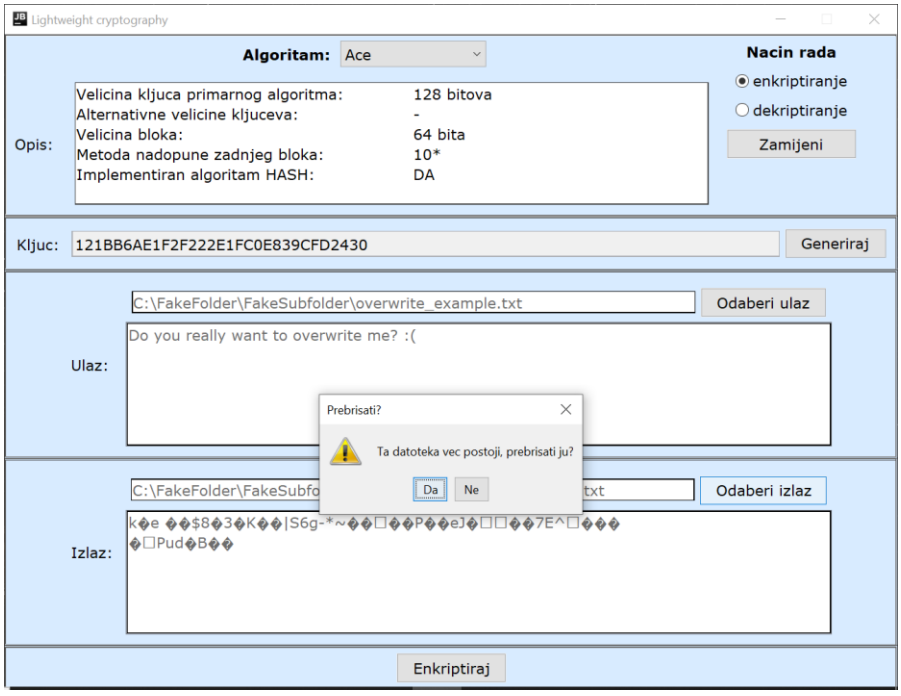
Slika 41. Biranje algoritma

Pri odabiru algoritma, prikazuju se neke osnovne informacije vezane za taj algoritam te se generira novi ključ. Odabirom ulazne datoteke prikazuje se sadržaj te datoteke i ažurira se ime izlazne datoteke dodavanjem "_encrypted" u slučaju enkriptiranja, ili "_decrypted" u slučaju dekriptiranja. Promjena načina rada također mijenja završetak imena izlazne datoteke ovisno o novoodabranom načinu rada, ali i mijenja tekst gumba za provođenje kriptiranja.
S obzirom da se izlazna datoteka stvara tek kad se stisne glavni gumb, a ne odmah pri odabiru datoteke, ako se ulaz pokuša zamijeniti izlazom koji još nije stvoren provođenjem kriptiranja biti će prikazan prozor koji javlja grešku.



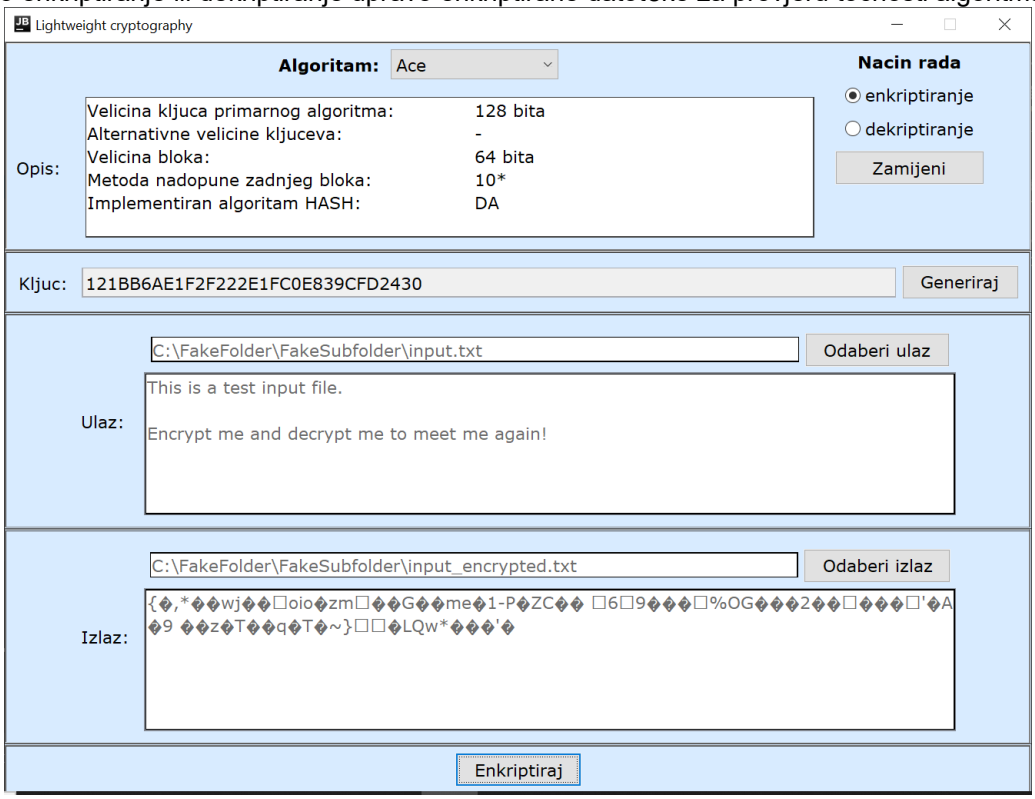
Slika 42. Greška ako datoteka još ne postoji

Ako se za izlaznu datoteku odabere datoteka koja već postoji, korisnika će se upozoriti na potrebu prebrisavanja iste.

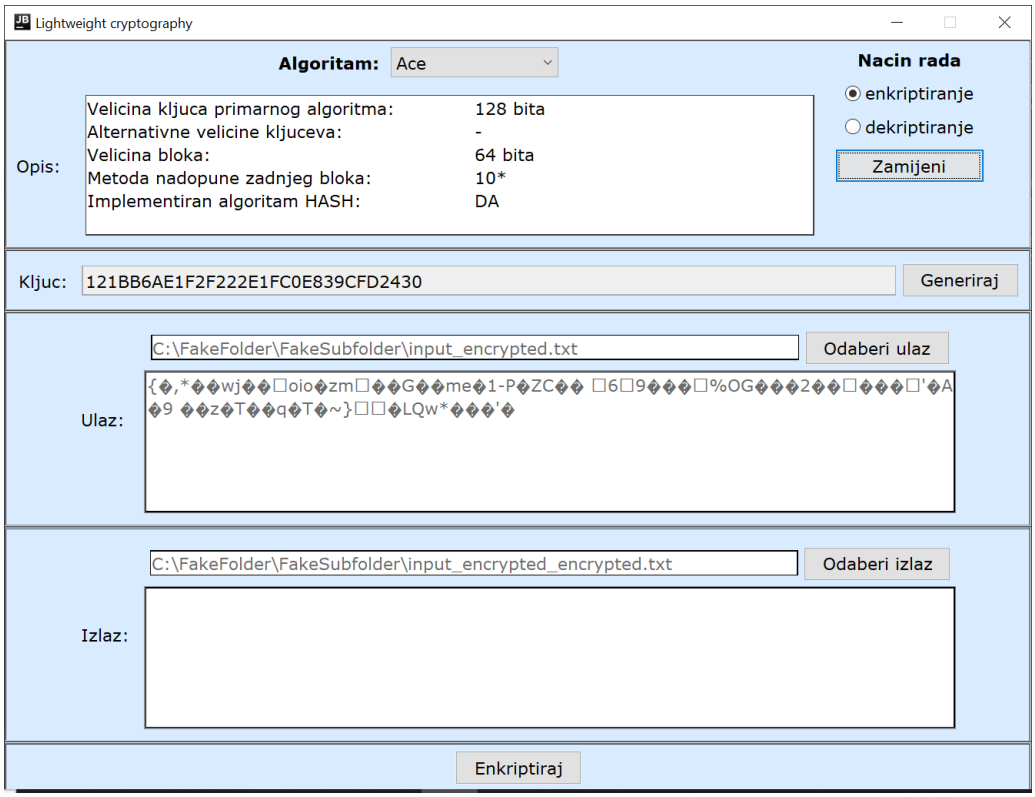


Slika 43. Upozorenje na prebrisavanje

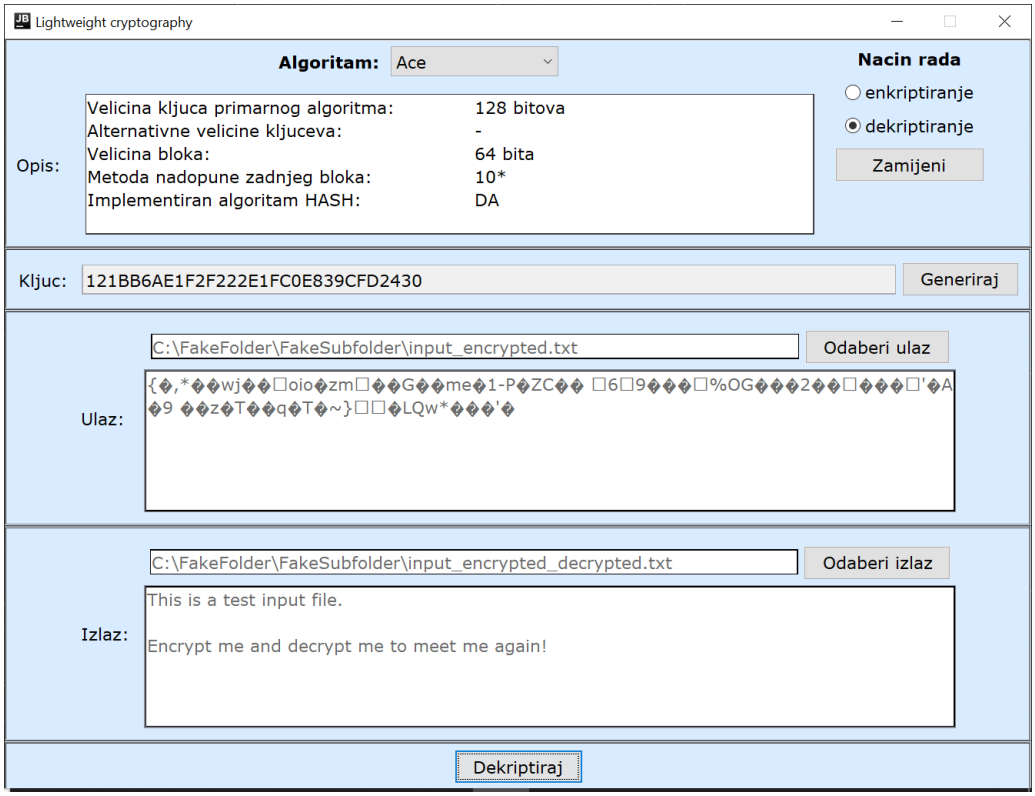
Sučelje nudi i gumb “Zamijeni” koji trenutnu izlaznu datoteku postavlja za ulaznu kako bi se olakšalo višestruko enkriptiranje ili dekriptiranje upravo enkriptirane datoteke za provjeru točnosti algoritma.



Slika 44. Enkriptiranje



Slika 45. Zamjena izlaza na ulaz



Slika 46. Dekriptiranje

5. Literatura

1. A. Chakraboti, N. Datta, A. Jha, M. Nandi, *HyENA*, 29 .ožujka 2019.
2. J. Daemen, S. Hoffert, M. Peeters, G.V. Assche, R. Van Keer, *Xoodyak a lightweight cryptographic scheme*, 29 .ožujka 2019.
3. M. Aagaard, R. AlTawy, G. Gong, K. Mandal, R. Rohit, *ACE: An Authenticated Encryption and Hash Algorithm*
4. T. Beyne, Y. L. Chen, C. Dobraunig, B. Mennink, *Elephant v1*
5. S. Banik ,A. Chakraborti ,T. Iwata ,K. Minematsu, M. Nandi, T. Peyrin ,Y. Sasaki, S. Meng Sim ,Y. Todo, *GIFT-COFB*
6. D. J. Bernstein, S. Kölbl, S. Lucks ,P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.X. Standaert, Y. Todo, B. Viguier, *Gimli*
7. M. Hell, T. Johansson, W. Meier, J. Sönnerup, H. Yoshida, *Grain-128AEAD - A lightweight AEAD stream cipher*
8. A. Chakraborti, N. Datta, A. Jha, C. M. Lopez, M.I Nandi, Y. Sasaki, *LOTUS-AEAD and LOCUS-AEAD*
9. B. Chakraborty, M. Nandi, *mixFeed,* 22. ožujka 2019.
10. B. Chakraborty, M. Nandi, *ORANGE,* 22. ožujka 2019.
11. Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, Kan Yasuda, *PHOTON-Beetle Authenticated Encryption and Hash Family,* 29 .ožujka 2019.
12. T. Iwata, M. Khairallah , K. Minematsu, T. Peyrin , *Romulus*
13. D. Bellizia,F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, O. Pereira, T. Peters, F. X. Standaert,F. Wiemer, *Spook: Sponge-Based Leakage-Resilient Authenticated Encryption with a Masked Tweakable Block Cipher,* 29 .ožujka 2019.
14. J. Daemen, P. M. C. Massolino, Y.Rotella, *The Subterranean 2.0 cipher suite,* 29 .ožujka 2019.
15. McGrew D., Siječanj 2008, *An interface and Algorithms for Authenticated Encryption,* dostupno na: https://tools.ietf.org/html/rfc5116

Svi radovi dostupni: https://csrc.nist.gov/projects/lightweight-cryptography/round-1-candidates