

Univerzitet Singidunum

**Master studijski program: Savremene informacione
tehnologije**

Nikola Pavličević

**Tema: Komparativna analiza novih hash funkcija
Master rad**

Mentor: Prof dr Mladen Veinović

Student: Nikola Pavličević, broj indeksa 410 394/2011

Sadržaj

Abstrakt.....	3
Abstract.....	3
Metodologija.....	4
Cilj.....	4
Predmet istraživanja.....	4
Metode istraživanja i tok istraživačkog procesa.....	5
Uvod.....	6
Hash funkcije.....	7
Svojstva sigurnih hash funkcija.....	7
Primene hash funkcija.....	8
Istorija hash funkcija.....	13
MD5 (Message digest 5).....	13
Blake.....	18
Dizajn funkcije.....	18
Pregled funkcije.....	18
Očekivana snaga.....	19
Prednosti i ograničenja.....	19
Detaljan dizajn.....	20
Optimizacije u ovom radu.....	24
Keccak.....	32
Sunder.....	32
Keccak permutacije.....	34
Opis rešenja.....	36
Blake klasa.....	36
Keccak klasa.....	37
Aplikacija.....	39
Rezultati.....	40
Zaključak.....	43

Abstrakt

Ovaj rad je efikasna objektno-orijentisana implementacija Blake i Keccak hash funkcija u jeziku C++. Ideja je da se implementacija može koristiti u raznim aplikacijama, za primer kontrola verzije softvera ili digitalni potpisi. Cilj nam je videti koja je od dve funkcije efikasnija kod računja sume od 256 bita. Našli smo Blake postiže manje ciklusa po bajtu od Keccak-a.

Ključne reči: hash, efikasnost, objektno-orijentisano

Abstract

This project is efficient object-oriented implementation of Blake and Keccak hash functions in C++. The idea is for implementation to be useful for various applications, for example software version control or digital signatures. The goal is to see which of the two functions is faster when computing sums of 256 bits. We found that Blake achieves less cycles per byte than Keccak.

Key words: hash, efficiency, object-oriented

Metodologija

Cilj

Cilj ovog projekta je efikasna objektno-orijentisana implementacija hash funkcija Blake i Keccak u jeziku C++. Da bi se ispitala efikasnost meriče se ciklusi po bajtu za poruke različitih veličina. Rezultati dve funkcije će se porediti.

Ovaj projekat je pokrenut iz želje da se napravi efikasna objektno-orijentisana implementacija novih hash funkcija zbog njihove sigurnosti, jednostavnosti i brzine. Ovo su veoma poželjne osobine u okruženju sa velikom količinom podataka ili velikim brojem transakcija u svakom trenutku. Sledeća želja je da se naprave implementacije sa čistim i razumljivim kodom koje se mogu koristiti u obrazovne svrhe.

Predmet istraživanja

Predmet istraživanja su kriptografske hash funkcije.

Metode istraživanja i tok istraživačkog procesa

Jedna od metoda u ovom projektu biće programiranje. Blake i Keccak algoritmi će se implementirati. Dalje će se meriti ciklusi po bajtu za poruke različitih veličina.

Više merenja će se obaviti da bi se dobio statistčki reprezentativan uzorak.

Sledeće će se pokazati koji od algoritama je efikasniji računanjem srednje vrednosti i standardne promene.

Uvod

Ovaj rad je efikasna objektno-orientisana implementacija novih hash funkcija Blake i Keccak u jeziku C++. Za tim imamo više potreba. Jedna je kontrola verzije softvera. Tipično ovde hashiramo nekoliko gigabajta da bi dobili trenutnu verziju. To radimo često. Potrebne su nam hash funkcije koje mogu brzo dobiti sumu i ne proizvode sudare.

Takođe su nam potrebne sigurnije hash funkcije. Za trenutne hash funkcije slabosti su poznate ili se naziru. Za nove još uvek ne. Za primer za Blake razlike su pronađene samo za 7 od 14 rundi. Slično je za Keccak, postoje razlike samo za smanjen broj rundi. Zanimljivo je napomenuti da Keccak ima veliku sigurnosnu marginu. Čak i ako smanjimo broj rundi sa 24 na 12 još uvek imamo sigurnu funkciju.

Takođe su nam potrebne hash funkcije koje se veoma lako i efikasno mogu paralelizovati. Nove hash funkcije su predviđene za ovo. Veoma su modularne i njihove operacije na blokovima se mogu raditi na više procesora istovremeno. To je veoma bitno kad imamo velik broj procesa u svakom trenutku.

U isto vreme kad paralelizujemo funkcije hoćemo da na kraju dobijemo tačan rezultat. To je još jedan razlog za izbor C++ a. Novi standard C++ 11 donosi mnogo novih stvari koje možemo iskoristiti za sigurno, jednostavno i brzo paralelizovanje. Drugim rečima kod sigurnog paralelizovanje možemo biti sigurni da u bilo kom trenutku samo jedan proces ima pristup jednom resursu. To možemo postići koristeći neku od biblioteka. Gledajući jednostavnost postoje izmene u sintaksi koje nam olakšavaju pisanje softvera i daju čitkiji izvor. Primer je automatska detekcija tipa promenljivih. Tip se detektuje na osnovu date vrednosti što nam ostavlja više vremena za pisanje izvora i logike. Manje moramo misliti na tipove i da li smo ih dobro dodelili. Gledajući brzinu programeri mogu brže naučiti jezik i početi implementirati algoritme.

Peta potreba je jednostavnost. Hoćemo jednostavne algoritme koje možemo brzo implementirati, proveriti i koji se sami po sebi brzo izvršavaju. Jednostavnost nam je potrebna, pored brzine implementacije, zbog razumevanja. Jednostavniji algoritam možemo lakše proveriti i videti koliko je bezbedan. Takođe ako hoćemo implementirati algoritam u hardveru rezultat je efikasniji i jeftiniji.

Hash funkcije

U najosnovnijem obliku hash funkcija je suma poruke. To za primer može biti suma heksadecimalnih vrednosti. Za primer ako imamo poruku 0x270x280x29 suma je 84. Problem u ovom slučaju je što možemo naći drugu kombinaciju sadržaja koja nam daje isti hash. Za primer 0x250x280x31. Drugim rečima isti hash dobijamo za više različitih poruka. Hoćemo nešto sigurnije, to jeste, da jedan hash odgovara samo jednoj poruci. Potrebno nam je ako hoćemo da proverimo integritet dobijene poruke.

Svojstva sigurnih hash funkcija

Hash funkcija za opštu upotrebu, za primer proveru integriteta poruke, je efikasna. Proizvoljnu poruku $\{0,1\}^n$, gde je n proizvoljan broj bitova, kompresuje u oblik $\{0,1\}^k$, gde je k određen broj bitova, brzo.

Dobra hash funkcija raspoređuje vrednosti uniformno. Verovatnoća da nasumična poruka da određenu sumu je $\frac{1}{2^k}$.

Sledeće teško je invertovati sumu. Drugim rečima ako imamo sumu teško je dobiti originalnu poruku iz nje. Ako uzmemo x za poruku a y za sumu teško je naći $h(x)=y$, gde je h hash funkcija.

Dalje teško je naći sudar. Ako gledamo slabu definiciju otpornosti i ako imamo poruku x_1 i sumu $h(x_1)$ teško je naći drugu poruku x_2 tako da njihove sume budu jednake, to jeste, da imamo $h(x_1)=h(x_2)$.

Ako gledamo jaču definiciju otpornosti teško nam je naći bilo koje dve poruke za koje bi sume bile iste. To jeste teško je naći bilo koje dve poruke x_1 i x_2 za koje imamo $h(x_1)=h(x_2)$.

Primene hash funkcija

Suma za proveru integriteta poruke (Hashed message authentication code, HMAC)

Svrha HMAC je očuvanje autentičnosti i integriteta poruke. HMAC koristi kriptografski ključ i sigurnu hash funkciju da dobije kod za proveru poruke (eng. message authentication code, MAC). MAC dobijamo pomoću tajnog ključa koji dele dve strane, pošiljaoc i primaoc. Pošiljaoc pošalje poruku zajedno sa MAC. Primaoc izračuna MAC iz poruke. Ako su dva MAC ista onda je poruka u redu, primljena je bez izmena.

Detalji HMAC algoritma

HMAC prima poruku i ključ i daje poruku i MAC. Na prvom nivou HMAC parser pravi ključ za hash funkciju. Ako nam je hash funkcija 256 bita ključ će nam biti iste dužine. Ako je ključ iste dužine kao hash ostavljamo ga nepromenjenog, ako je kraći dopunjujemo ga do dužine hash-a nulama i ako je duži od hash-a hashiramo ga i rezultat nam je ključ. Nakon ovoga trebaju nam četiri dela, $X = K_0 \oplus OPAD$, $Y = K_0 \oplus IPAD$, $Z = H(Y + M)$ i $A = H(X + Z)$, gde je OPAD 0x36 dužine hash-a, IPAD 0x5c dužine hash-a i + spajanje nizova. MAC nam je t najvažnijih bajtova od A.

Uzmimo za primer jednu osobu zvanu Bob. Njoj su data dva ključa, jedan privatni i drugi javni. Samo Bobu je poznat njegov privatni ključ dok on daje na neki siguran način svoj javni ključ svima sa kojim hoće da komunicira. Ako Bob hoće da pošalje poruku drugoj osobi, za primer Klari, ona može izračunati sažetak poruke, bitove koji su jedinstveni određenoj poruci, koristeći određen algoritam, Blake ili Keccak u našem slučaju, i šifrovati ga svojim privatnim ključem. Kad Klara dobije poruku i potpis ona može dešifrovati poruku koristeći Bobov javni ključ. Ako je dešifrovanje uspešno Klara zna da je Bob poslao poruku. Sledeće Klara može izračunati sažetak poruke. Ako je njen sažetak jednak Bobovom onda ona zna da poruka nije izmenjena. Ono što je Bob poslao Klari uz poruku nazivamo digitalnim potpisom. Dakle digitalni potpis je sažetak poruke šifrovan privatnim ključem i koristi se za proveru identiteta pošiljaoca i integriteta poruke.

Formalno digitalni potpis je uređena trojka algoritama (Gen, Sign, Vrfy) koja zadovoljava sledeće uslove:

Algoritam za generisanje ključeva Gen uzima n bitova i daje par ključeva (pk, sk) , gde je pk javni ključ a sk tajni ključ.

Algoritam za potpisivanje Sign uzima za ulaz tajni ključ sk i poruku m gde $m \in \{0,1\}^*$. Na izlazu daje potpis s . Dužina s zavisi od algoritma za potpisivanje.

Deterministički algoritam Vrfy uzima za ulaz javni ključ pk , poruku m i potpis s . Na izlazu daje bit b , gde $b=1$ znači validan potpis a $bit=0$ znači nevalidan potpis.

Znak bita zavisi od dve operacije, mogućnosti dešifrovanja potpisa i provere sažetka poruke. $bit=0$ će biti ako nismo u stanju da potpis dešifrujemo ili ako nismo dobili isti sažetak kao i pošiljalac.

Čemu služe digitalni potpisi? Jedna od primena je provera identiteta. Pošto su privatni ključevi jedinstveni za svaku osobu znamo koja je osoba poslala poruku.

Dalje digitalni potpisi se koriste za održanje integriteta poruke, što je već prikazano na početku. Nemogućnost poricanja slanja poruke je još jedna primena. Pošto je privatni ključ vezan za određenu osobu ona ne može poreći da je poslala poruku.

Potpisi sa RSA algoritmom

Generisanje ključeva

Prvo se izaberu dva velika prosta broja, p i q , često nekoliko stotina bitova u dužini.

Izračuna se $n=pq$.

Izračuna se $r=(p-1)(q-1)$.

Izabere se javni eksponent e takav da su on i r uzajamno prosti.

Izračuna se privatni eksponent $d=e^{-1}(\text{mod } r)$.

Javni ključ je (n,e) .

Privatni ključ je d .

Šifrovanje

$C=m^e(\text{mod } n)$ je šifrovana poruka

Dešifrovanje

$m=C^d(\text{mod } n)$ je poruka.

Potpisi sa eliptičnim krivama (ECDSA)

Da bi Alisa (A) poslala potpisanu poruku Bobu (B) oni se moraju dogovoriti oko veličine polja, parametara a i b za krivu $y^2=x^3+ax+b$ i generatorsku tačku G veličine n . Alisa takođe mora izabrati d_a takvo da $d \in [1, n-1]$ i izračunati javni ključ Q_a , $Q_a=d_a G$.

Da bi potpisala poruku Alisa prati sledeći algoritam:

Izračuna $e=HASH(m)$ gde je $HASH$ arbitrarna kriptografski sigurna funkcija za dobijanje sažetka poruke.

Neka je l_0 prvi bit sa leve strane od e .

Bira nasumičan broj k takav da $k \in [1, n-1]$.

Izračuna $r=x_1(\text{mod } n)$ gde je $(x_1, y_1)=kG$. Ako je $r=0$ vraća se na predhodni korak.

Izračuna $s = k^{-1}(i_0 + rd_a)(\text{mod } n)$. Ako je $s=0$ vraća se na treći korak.

Potpis je (r,s) .

Da bi Bob proverio potpis mora imati kopiju Q_a i pratiti algoritam:

Izračuna $e = \text{HASH}(m)$ gde je HASH arbitrarna kriptografski sigurna funkcija za dobijanje sažetka poruke.

Neka je l_0 prvi bit sa leve strane od e .

Izračuna $w = s^{-1}(\text{mod } n)$.

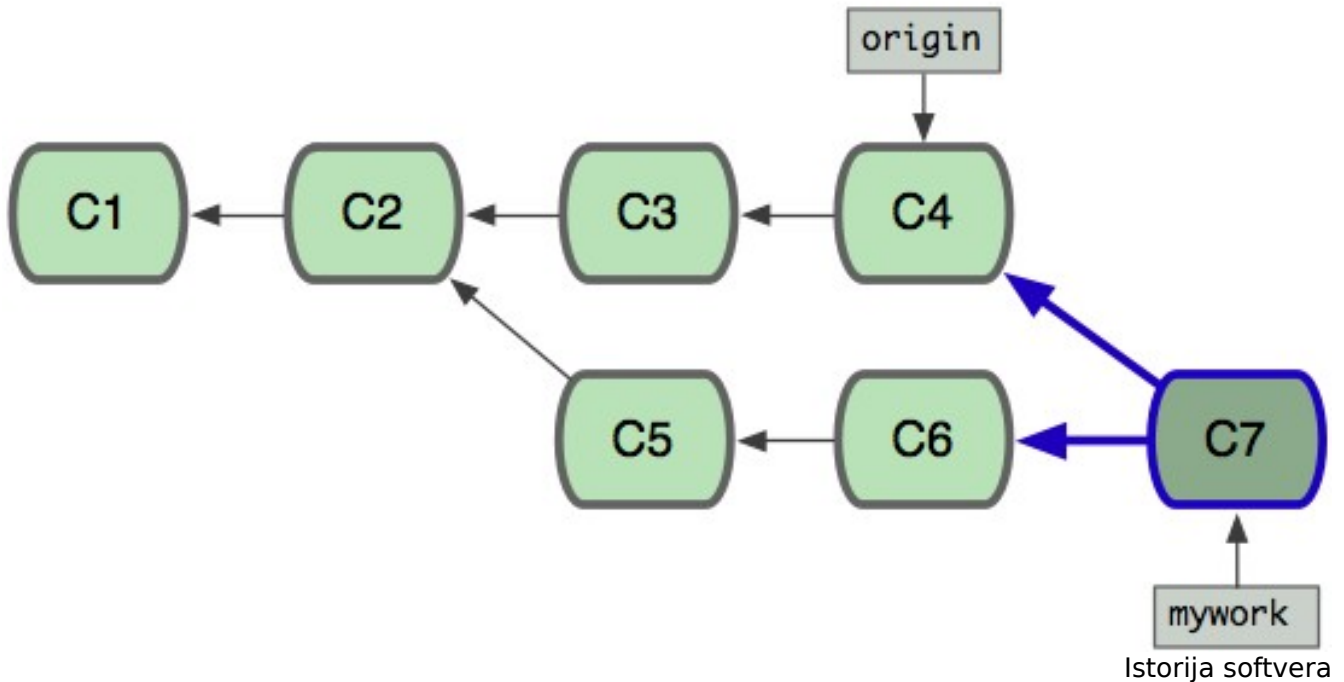
Izračuna $u_1 = I_0 w(\text{mod } n)$ i $u_2 = rw(\text{mod } n)$.

Izračuna $(x_1, y_1) = u_1 G + u_2 Q_a$.

Potpis je validan ako je $r = x_1(\text{mod } n)$.

Kontrola verzije softvera je čuvanje sadašnjeg stanja softvera kao i prošlih. Dozvoljava veću produktivnost kad se koristi u timu. Svaki član može dodati svoje izmene istovremeno. Mnogo brže nego kad se radi u nizu. Istoriju softvera možemo prikazati grafom kao u sledećoj slici.

git merge



Istorija hash funkcija

MD5 (Message digest 5)

MD5 je kriptografska hash funkcija koja daje sume od 128 bita. Napravio je Ron Rivest 1991. da zameni predhodnu verziju. MD5 suma se tipično prikazuje kao heksadecimalni broj dužine 32 cifre.

Algoritam

MD5 računa sumu od 128 bita od poruke bilo koje dužine. Poruku delimo u blokove od 512 bita. Ako je potrebno poruku proširimo tako da njena dužina u bitovima bude deljiva sa 512. Kad proširujemo prvo dodamo bit 1 zatim 0 koliko nam je potrebno da dužina bude 64 bita manja od umnoška 512. Zatim dodamo 64 bitnu dužinu poruke.

Glavni MD5 algoritam radi na stanju od 128 bita podeljeno u četiri 32 bitne reči A, B, C i D. U početku one sadrže određene konstante. Glavni algoritam radi sa 512 bitnim blokovima menjajući stanje. Obrada bloka se sastoji od četiri sličnih stavki koje se zovu runde. Svaka runda se sastoji od 16 sličnih operacija koje su nelinearna funkcija F, modularno sabiranje i rotacija. Postoje četiri različite funkcije

F, po jedna za svaku rundu. To su $F(X,Y,Z)=(X\wedge Y)\vee(\neg X\wedge Z)$,
 $G(X,Y,Z)=(X\wedge Z)\vee(Y\wedge\neg Z)$, $H(X,Y,Z)=X\oplus Y\oplus Z$ i $I(X,Y,Z)=Y\oplus(X\vee\neg Z)$.

Pseudokod

//Note: All variables are unsigned 32 bit and wrap modulo 2^{32} when calculating

```
var int[64] r, k
```

//r specifies the per-round shift amounts

```
r[ 0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}
```

```
r[16..31] := {5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20}
```

```
r[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}
```

```
r[48..63] := {6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21}
```

//Use binary integer part of the sines of integers (Radians) as constants:

```
for i from 0 to 63
```

```
    k[i] := floor(abs(sin(i + 1)) × (2 pow 32))
```

```
end for
```

//(Or just use the following table):

```
k[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
```

```
k[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
```

```
k[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
```

```
k[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
```

```
k[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
```

```
k[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
```

```
k[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
```

```
k[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
```

```
k[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
```

```
k[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
```

```
k[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
```

```
k[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
```

```
k[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
```

```
k[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
```

```
k[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
```

```
k[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }
```

```
//Initialize variables:
```

```
var int h0 := 0x67452301 //A
```

```
var int h1 := 0xefcdab89 //B
```

```
var int h2 := 0x98badcfe //C
```

```
var int h3 := 0x10325476 //D
```

```
//Pre-processing: adding a single 1 bit
```

```
append "1" bit to message
```

```
/* Notice: the input bytes are considered as bits strings,
```

```
where the first bit is the most significant bit of the byte.
```

```
//Pre-processing: padding with zeros
```

```
append "0" bit until message length in bit  $\equiv 448 \pmod{512}$ 
```

```
append length mod (2 pow 64) to message
```

```
//Process the message in successive 512-bit chunks:
```

```
for each 512-bit chunk of message
```

```
break chunk into sixteen 32-bit words  $w[j]$ ,  $0 \leq j \leq 15$ 
```

```
//Initialize hash value for this chunk:
```

```
var int a := h0
```

```
var int b := h1
```

```
var int c := h2
```

```
var int d := h3
```

```
//Main loop:
```

```
for i from 0 to 63
```

```
if  $0 \leq i \leq 15$  then
```

```
f := (b and c) or ((not b) and d)
```

```
g := i
```

```
else if  $16 \leq i \leq 31$ 
```

```

f := (d and b) or ((not d) and c)

g := (5×i + 1) mod 16

else if  $32 \leq i \leq 47$ 

f := b xor c xor d

g := (3×i + 5) mod 16

else if  $48 \leq i \leq 63$ 

f := c xor (b or (not d))

g := (7×i) mod 16

temp := d

d := c

c := b

b := b + leftrotate((a + f + k[i] + w[g]) , r[i])

a := temp

end for

//Add this chunk's hash to result so far:

h0 := h0 + a

h1 := h1 + b

h2 := h2 + c

h3 := h3 + d

end for

var char digest[16] := h0 append h1 append h2 append h3  //(Output is in little-endian)

//leftrotate function definition

leftrotate (x, c)

return (x << c) binary or (x >> (32-c));

```

Primer sume

MD5("") = d41d8cd98f00b204e9800998ecf8427e

Slabosti

1993. Den Ber i Boselers su našli pseudo sudar za MD5, to jeste, dva inicijalizaciona vektora koji su davali istu sumu. 1996. Dobertin je dobio sudar za MD5 funkciju. I ako nije bio za pun MD5 bio je povod da počnemo koristiti druge algoritme. 17. Avgusta 2004. su nađeni sudari za pun MD5.

Blake

Blake [1] je kriptografska hash funkcija koju su napravili Žan Filip Omason, Luka Henzen, Vili Meier i Rafael Fan. Bazirana je na prehodno dobro proučenim komponentama izabranim na osnovu svoje komplementarnosti. Nasleđe Blake-a je trostruko. Za iteracije koristi HAIFA (hash iterative framework) od Bihama i Dankelmana. U unutrašnjosti Blake koristi širok prostor zbog čega lokalni sudari nisu mogući. To je rezultat koji se ne oslanja na neefikasnost računanja nečega. Algoritam za kompresiju je modifikovana verzija ChaCha šifre čija je sigurnost dobro poznata i performansa odlična. Algoritam je isto veoma lak za paralelizovanje.

Dizajn funkcije

Funkcija daje sume od 224, 256, 384 i 512 bita. Maksimalna dužina poruke je $2^{64}-1$ bita. U isto vreme Blake može hashirati sa solju. Podoban je za lake implementacije.

Pregled funkcije

Algoritam	Dužina reči u bitima	Dužina poruke u bitima	Dužina bloka u bitima	Dužina sume u bitima	Dužina soli u bitima	
224	32	<264	512	224	128	
256	32	<264	512	256	128	
384	64	<2128	1024	384	256	
512	64	<2128	1024	512	256	

Blake prati HAIFA iteraciju. Funkcija za kompresiju zavisi od soli i broja bitova obrađenih do sada, brojača. Kod kompresije veliko unutrašnje stanje se inicijalizuje početnim vrednostima, solju i brojačem. Onda se osvežava rundama zavisnim od poruke i na kraju se kompresuje da bi dobili sledeće vrednosti za ulančavanje. Unutrašnje stanje funkcije za kompresiju se prikazuje kao 4×4 matrica reči.

Runda funkcije je modifikovana ChaCha šifra duple runde. Prvo se sve četiri kolone osvežavaju nezavisno i onda četiri diagonale. Kad osvežavamo svaku kolonu i diagonalu uzimamo dve reči poruke prema permutaciji zavisnoj o rundi. Posle niza rundi stanje smanjujemo na polovinu dužine.

Očekivana snaga

Najjači napad na funkciju bi trebao biti sirova sila za pronalaženje sudara, sa istom ili različitom solju. Trebalo bi biti nemoguće razlikovati Blake sa nasumičnom solju od nasumičnog niza ako imamo pristup funkciji kao crnoj kutiji. Tačnije treba nam barem 2^s upita kutiji, gde je s dužina soli u bitima.

Prednosti i ograničenja

Prednosti

Dizajn

Jednostavnost algoritma

Interfejs hashiranja sa solju

Performansa

Brza u hardveru i softveru.

Sigurnost

Zasnovana na veoma poznatoj komponenti ChaCha.

Otporna na uobičajene napade gledajući sudare.

Otporna na posmatranje (side channel attacks).

Otporna na povećanje dužine.

Ograničenja

Dužina poruke ograničena na 2^{64} i 2^{128} bita za Blake 256 i Blake 512.

Iste tačke, gde je $Hash(X)=X$ su pronađene za manje vremena nego za idealnu funkciju ali ne efikasno.

Detaljan dizajn

Početni hash

IV0= 6A09E667 IV1= BB67AE85
IV2= 3C6EF372 IV3= A54FF53A
IV4= 510E527F IV5= 9B05688C
IV6= 1F83D9AB IV7= 5BE0CD19

Konstante

c0= 243F6A88 c1= 85A308D3
c2= 13198A2E c3= 03707344
c4= A4093822 c5= 299F31D0
c6= 082EFA98 c7= EC4E6C89
c8= 452821E6 c9= 38D01377
c10= BE5466CF c11= 34E90C6C
c12= C0AC29B7 c13= C97C50DD
c14= 3F84D5B5 c15= B5470917

Ovo su prve cifre π .

Permutacije koje koristi funkcija

σ_0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 σ_1 14 10 4 8 9 15 13 6 1 12 0 2 11 7 5 3
 σ_2 11 8 12 0 5 2 15 13 10 14 3 6 7 1 9 4
 σ_3 7 9 3 1 13 12 11 14 2 6 5 10 4 0 15 8
 σ_4 9 0 5 7 2 4 10 15 14 1 11 12 6 8 3 13
 σ_5 2 12 6 10 0 11 8 3 4 13 7 5 15 14 1 9
 σ_6 12 5 1 15 14 13 4 10 0 7 6 3 9 2 8 11
 σ_7 13 11 7 14 12 1 3 9 5 0 15 4 8 6 2 10
 σ_8 6 15 14 9 11 3 0 8 12 2 13 7 1 4 10 5
 σ_9 10 2 8 4 7 6 1 5 15 11 9 14 3 12 13 0

Ulaz funkcije

Lanac $h = h_0, \dots, h_7$.

Blok poruke $m = m_0, \dots, m_{15}$.

So $s = s_0, \dots, s_3$.

Brojač $t = t_0, t_1$.

Četiri ulaza su 30 reči ukupno. Izlaz je novi lanac $h' = h'_0, \dots, h'_7$. Kompresiju možemo napisati kao $h' = \text{compress}(h, m, s, t)$.

Inicijalizacija

$$\begin{pmatrix} v_0 v_1 v_2 v_3 \\ v_4 v_5 v_6 v_7 \\ v_8 v_9 v_{10} v_{11} \\ v_{13} v_{14} v_{15} v_{16} \end{pmatrix} = \begin{pmatrix} h_0 h_1 h_2 h_3 \\ h_4 h_5 h_6 h_7 \\ s_0 \oplus c_0 s_1 \oplus c_1 s_2 \oplus c_2 s_3 \oplus c_3 \\ t_0 \oplus c_4 t_0 \oplus c_5 t_1 \oplus c_6 t_1 \oplus c_7 \end{pmatrix}$$

Funkcija runde

Posle inicijalizacije funkcija runde se ponavlja 14 puta i kompresuje ulaz. U svakom krugu računamo $G_0(v_0, v_4, v_8, v_{12})$, $G_1(v_1, v_5, v_9, v_{13})$, $G_2(v_2, v_6, v_{10}, v_{14})$, $G_3(v_3, v_7, v_{11}, v_{15})$,

$G_4(v_0, v_5, v_{10}, v_{15})$, $G_5(v_1, v_6, v_{11}, v_{12})$, $G_6(v_2, v_7, v_8, v_{13})$, $G_7(v_3, v_4, v_9, v_{14})$.
 $G_i(a, b, c, d)$ je

$$\begin{aligned} a &\leftarrow a + b + (m_{or(2i)} \oplus c_{or(2i+1)}) \\ d &\leftarrow (d \oplus a) \text{ror } 16 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \text{ror } 12 \\ a &\leftarrow a + b + (m_{or(2i+1)} \oplus c_{or(2i)}) \\ d &\leftarrow (d \oplus a) \text{ror } 8 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \text{ror } 7 \end{aligned}$$

i ror rotacija bita udesno. Prva četiri poziva $G_0, \dots, G_4$ mogu se računati paralelno zato što osvežavaju različite kolone. Slično četiri poslednja poziva $G_4, \dots, G_7$ se mogu paralelizovati. Kad smo premašili devetu rundu koristimo permutaciju $r \bmod 10$. Za primer ako smo u dvanaestoj rundi koristimo drugu permutaciju, $12 \bmod 10 = 2$.

Finalizacija

Posle četrnaest rundi imamo nove lančane vrednosti

$$\begin{aligned} h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\ h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\ h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\ h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\ h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\ h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\ h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\ h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15} \end{aligned}$$

Računanje sume

Prvo proširimo poruku tako da njena dužina bude kongruentna 447 modulo 512. Proširujemo tako što dodamo bit 1 i zatim potreban broj nula. Dodaje se najmanje jedan bit a najviše 512. Zatim dodamo 64 bita koji predstavljaju dužinu poruke u bitima. Proširivanje možemo pokazati kao $m \leftarrow m + 1000\dots0001 + l_{64}$. Ovo nam garantuje da je dužina poruke umnožak 512 bita.

Zatim podelimo poruku u blokove dužine 16 reči. Sumu proširene poruke računamo kao

$$\begin{aligned} h^0 &\leftarrow IV \\ \text{for } i &= 0, \dots, N-1 \\ h^{i+1} &\leftarrow \text{compress}(h^i, m^i, s, l^i) \\ \text{return } &h^N \end{aligned}$$

Optimizacije u ovom radu

Permutacije

Permutacije smo izračunali unapred da ne bi morali tražiti ostatak u kasnijim rundama. Jednostavno korišćenje već postojećih vrednosti je mnogo brže od računanja. Umesto tabele permutacija prikazane ranije imamo tabelu

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3

11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4

7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8

9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13

2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9

12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11

13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10

6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5

10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3

11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4

7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8

Binarne operacije

Funkcija radi sa bitima. Zbog ovoga smo mogli zameniti operacije kao što su množenje i deljenje pomeranjem bita. Pomeranje ulevo predstavlja množenje. Za primer ako se pomerimo za 3 bita ulevo pomnožili smo sa 8, $2^3=8$. Pomeranje udesno predstavlja deljenje sa nekim stepenom 2, za primer, ako pomerimo udesno za dva bita podelili smo sa 4, $2^2=4$.

Primer u implementaciji, brojač bita

```
void Blake::update_hash_block(unsigned char * input,unsigned int
                                input_len_bytes)
{
    message_length_bits+=(input_len_bytes<<3);

    block_operate(input);
}
```

Ovde računamo ukupan broj obrađenih bita. Kad dobijemo broj bajtova poruke, pomerimo ga ulevo za 3 mesta, to jeste, pomnožimo sa 8 i dobijemo bite. Razlog ovome je brzina. Pomeranje bita je mnogo brže od množenja ili deljenja.

Proširenje poruke

Pošto radimo sa bajtovima postoje posebni slučajevi koje koristimo kod proširenja poruke tako da ne moramo računati broj bita koji se mora dodati. Prvi slučaj je kad je poslednji blok 440 bita ili 55 bajtova. Tada samo dodajemo bite 0x81 na kraj bloka i zatim 64 bita dužine poruke. U implementaciji imamo

```
        input[input_len_bytes]=0x81;

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>56);

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>48);

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>40);

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>32);

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>24);

        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>16);

        input[++input_len_bytes]=(unsigned char)(message_length_bits>>8);

        input[++input_len_bytes]=(unsigned char)(message_length_bits);

        block_operate(input);
```

Drugi slučaj je poslednji blok manji od 440 bita. Tada dodajemo bite 0x80 na kraj poruke. Poslednja 72 bita su nam 0x01 i 64 bita dužine poruke. Između imamo nule. U implementaciji imamo

```
        input[input_len_bytes++] = 0x80;

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 8);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 16);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 24);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 32);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 40);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 48);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 56);

        input[--block_size_temp_bytes] = 0x01;

    memset(input + input_len_bytes, 0, block_size_temp_bytes - input_len_bytes);

    block_operate(input);
```

U trećem slučaju imamo poslednji blok duži od 440 bita. U tom slučaju na kraj poslednjeg bloka dodajemo bite 0x80 i nule do kraja. U drugi blok stavljamo nule, bite 0x01 i 64 bita dužine poruke. U implementaciji imamo

```
input[input_len_bytes++] = 0x80;

memset(input + input_len_bytes, 0, block_size_bytes - input_len_bytes);

block_operate(input);

unsigned char buffer[block_size_bytes];

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 8);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 16);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 24);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 32);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 40);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 48);

buffer[--block_size_temp_bytes] = (unsigned char)
    (message_length_bits >> 56);

buffer[--block_size_temp_bytes] = 0x01;

block_operate(buffer);
```

Cela funkcija za računanje konačne sume

```
void Blake::final_hash_block(unsigned char * input, unsigned int input_len_bytes)
{
    unsigned int block_size_temp_bytes=block_size_bytes;
    message_length_bits+=(input_len_bytes<3);
    if(input_len_bytes==55)
    {
        input[input_len_bytes]=0x81;
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>56);
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>48);
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>40);
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>32);
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>24);
        input[++input_len_bytes]=(unsigned char)
            (message_length_bits>>16);
        input[++input_len_bytes]=(unsigned char)(message_length_bits>>8);
        input[++input_len_bytes]=(unsigned char)(message_length_bits);
        block_operate(input);
    }
    else if(input_len_bytes<55)
    {
```

```

        input[input_len_bytes++] = 0x80;

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 8);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 16);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 24);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 32);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 40);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 48);

        input[--block_size_temp_bytes] = (unsigned char)
            (message_length_bits >> 56);

        input[--block_size_temp_bytes] = 0x01;

        memset(input+input_len_bytes, 0, block_size_temp_bytes-
            input_len_bytes);

        block_operate(input);

    }

    else

    {

        input[input_len_bytes++] = 0x80;

        memset(input+input_len_bytes, 0, block_size_bytes-input_len_bytes);

        block_operate(input);

```

```

        unsigned char buffer[block_size_bytes];

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>8);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>16);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>24);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>32);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>40);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>48);

        buffer[--block_size_temp_bytes]=(unsigned char)
            (message_length_bits>>56);

        buffer[--block_size_temp_bytes]=0x01;

        block_operate(buffer);

    }

}

```

Keccak

Keccak [2] je kriptografska hash funkcija koju su napravili Guido Bertoni, Džon Dejmen i Majkl Pitera. Po autorima funkcija bi trebala biti veoma brza.

Keccak koristi sunderastu konstrukciju gde blokove poruke upijamo u stanje pomoću XOR. Posle stanje nepovratno permutujemo. Stanje najviše može imati 1600 bita. Nakon poslednjeg bloka iscedimo sumu potrebne dužine.

Sunder

Sunder je klasa algoritama sa ograničenim unutrašnjim stanjem koji uzimaju niz bita bilo koje dužine i daju izlaz željene dužine.

Konstrukcija

Trebaju nam tri komponente za sunder

Stanje koje sadrži b bita

Funkcija f ograničene dužine koja permutuje stanje

Funkcija za proširenje P

Stanje je podeljeno u dva dela, R veličine r bita i C dužine $c=b-r$ bita. r nam je brzina i c kapacitet. Funkcija za proširenje dodaje bita koliko je potrebno da poruka bude umnožak r . Tada se poruka može podeliti u blokove od r bita.

Način rada

Stanje se postavi na nulu

Poruka se proširi

Prvi blok poruke XOR R

S zamenimo sa $f(S)$

Sledeći blok poruke XOR R

S zamenimo sa $f(S)$

.

.

.

Operacije ponavljamo za preostale blokove poruke dok ih sve ne upijemo.

Sad možemo iscediti sunder

R deo stanja su prvih r bita izlaza

S zamenimo sa $f(S)$

R deo stanja su drugih r bita izlaza

.

.

.

Isto ponavljamo dok ne dobijemo dužinu izlaza koju hoćemo

Primetimo da poruku nikada ne upijamo u C deo niti dobijamo izlaz iz C. Koliko menjamo C zavisi od funkcije f. Kod hash funkcija otpornost na sudare zavisi od C i njegova dužine je tipično dva puta veća od nivoa otpornosti koji hoćemo.

Keccak permutacije

Postoje 7 Keccak permutacija koje obeležavamo sa Keccak-f(b) gde je $b=25 \times 2^l$. Keccak-f(b) je permutacija nad binarnim poljem gde su biti stanja označeni od 0 do b-1. B je širina permutacije.

Permutaciju Keccak-f(b) opisujemo kao niz operacija na stanju a koje je tridimenzionalni niz elemenata binarnog polja, to jeste $a[5][5][w]$. Izraz $a[x][y][z]$ predstavlja poziciju bita (x,y,z). Mapiranje između s i a je $s[w(5y + x) + z] = a[x][y][z]$.

Keccak-f(b) je permutacija sa ponavljanjem i sastoji se od niza r rundi. Runda se sastoji od pet koraka:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

$$\begin{aligned} \theta : a[x][y][z] &\leftarrow a[x][y][z] + \sum_{y'=0}^4 a[x-1][y'][z] + \sum_{y'=0}^4 a[x+1][y'][z-1] \\ \rho : a[x][y][z] &\leftarrow a[x][y][z - (t+1)(t+2)/2] \end{aligned}$$

gde je $t \quad 0 \leq t < 24$ i $\begin{pmatrix} 01 \\ 23 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ u $GF(5)$ ili $t = -1$ kad su $x = y = 1$

$$\begin{aligned} \pi : a[x][y] &\leftarrow a[x'][y'] \\ \chi : a[x] &\leftarrow a[x] + (a[x+1] + 1)a[x+2], \text{ sa } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 01 \\ 23 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \\ \iota : a &\leftarrow a + RC[i_r] \end{aligned}$$

Sabiranja i množenja su u $GF(2)$.

Pseudokod θ

```

for x=0 until 4 do
  C[x]=a[x,0]
for y=1 until 4 do
  C[x]=C[x]⊕a[x,y]
end for
end for
for x=0 until 4 do
  D[x]=C[x-1]⊕ROT(C[x+1],1)
for y=0 until 4 do
  A[x,y]=a[x,y]⊕D[x]
end for
end for

```

Pseudokod ρ

```
A[0,0]=a[0,0]

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

for t=0 until 23 do
A[x,y]=ROT(a(x,y),(t+1)(t+2)/2)

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

end for
```

Pseudokod π

```
for x=0 until 4 do
for y=0 until 4

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

A[X,Y]=a[x,y]
end for
end for
```

Pseudokod χ

```
for y=0 until 4 do
for x=0 until 4 do
A[x,y]=a[x,y]  $\oplus$  ((NOT a[x+1,y])  $\wedge$  a[x+2,y])
end for
end for
```

Opis rešenja

Imamo dve klase, Blake i Keccak, po jednu za svaku hash funkciju. Klase imaju statičke funkcije i promenljive, one koje su zajedničke svakom objektu. Takođe imamo funkcije koje su svojstvene svakom objektu.

Blake klasa

```
#include <iostream>

#include <cstring>

class Blake
{
private:
    unsigned int word[16]; //size in words
    static unsigned int p[14][16];
    unsigned long long t;
    unsigned int v[16]; //size in words
    unsigned int h[8]; //size in words
    static const unsigned int block_size_bytes;
    static unsigned int s[4]; //size in words
    static unsigned int cons[16]; //size in words

    unsigned long long message_length_bits;

    void get_word(unsigned char * block);
    unsigned int rotate_right(unsigned int x, unsigned int p);
```

```

        void block_operate(unsigned char * block);

public:
        Blake();
        ~Blake();

        void update_hash_block(unsigned char * input,unsigned int
                                input_len_bytes);
void final_hash_block(unsigned char * input, unsigned int input_len_bytes);
        void get_hash_bytes(unsigned char*result);
        unsigned int get_block_size_bytes();
};

```

Keccak klasa

```

        #ifndef KECCAK_H
        #define KECCAK_H

        #include <stdint.h>
        #include <string.h>

        #ifndef KECCAK_ROUNDS
        #define KECCAK_ROUNDS 24
        #endif

```

```

                                                                    #ifndef ROTL64
#define ROTL64(x, y) (((x) << (y)) | ((x) >> (64 - (y))))
                                                                    #endif

                                                                    class Keccak
                                                                    {
                                                                    private:
                                                                    static const uint64_t keccakf_rndc[24];
                                                                    static const int keccakf_rotc[24];
                                                                    static const int keccakf_piln[24];
                                                                    uint64_t state[25];
                                                                    static const unsigned int block_size_bytes;

                                                                    public:
                                                                    Keccak();

                                                                    // update the state
                                                                    void keccakf(uint64_t st[25], int norounds);

                                                                    // compute a keccak hash (md) of given byte length from "in"
                                                                    int keccak_hash_block(const uint8_t *in,int inlen,int mdlen);

                                                                    // compute final hash
                                                                    int keccak_final(

```

```

const uint8_t *in,
               int inlen,
               int mdlen
               );

void get_hash_bytes(uint8_t*hash,int hashlen);

unsigned int get_block_size_bytes(){
    return block_size_bytes;
}

};

#endif

```

Aplikacija

Komande aplikacije su date ispod.

`./main <hash> <testfile> <resultfile> <runs> <samples>`

`<hash>` - hash koji koristimo

`<testfile>` - fajl koji hashiramo

`<resultfile>` - fajl gde se upisuju rezultati (vreme hashiranja)

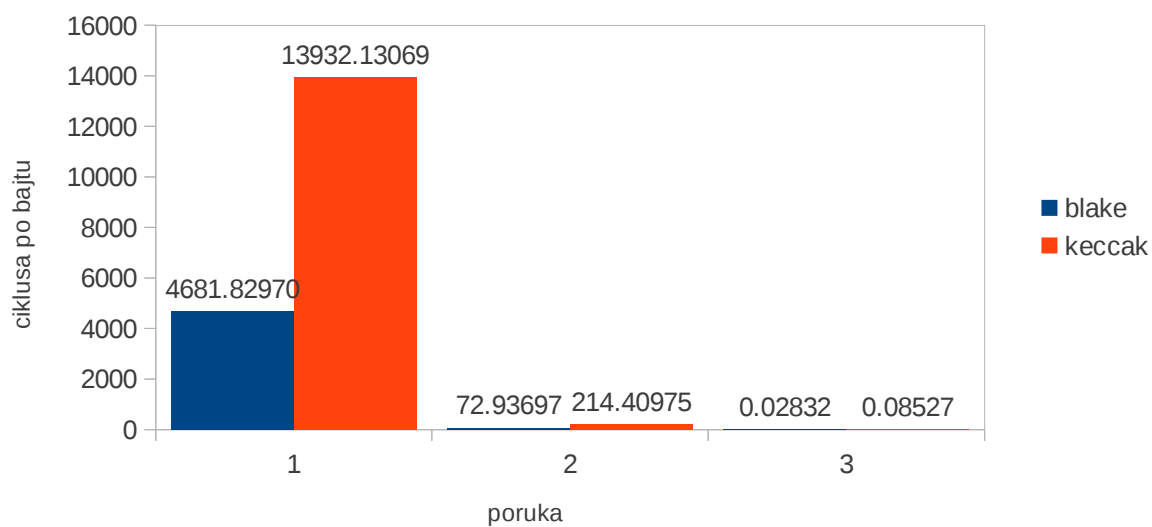
`<runs>` - koliko puta uzimamo uzorke

`<samples>` - koliko uzoraka uzimamo

Rezultati

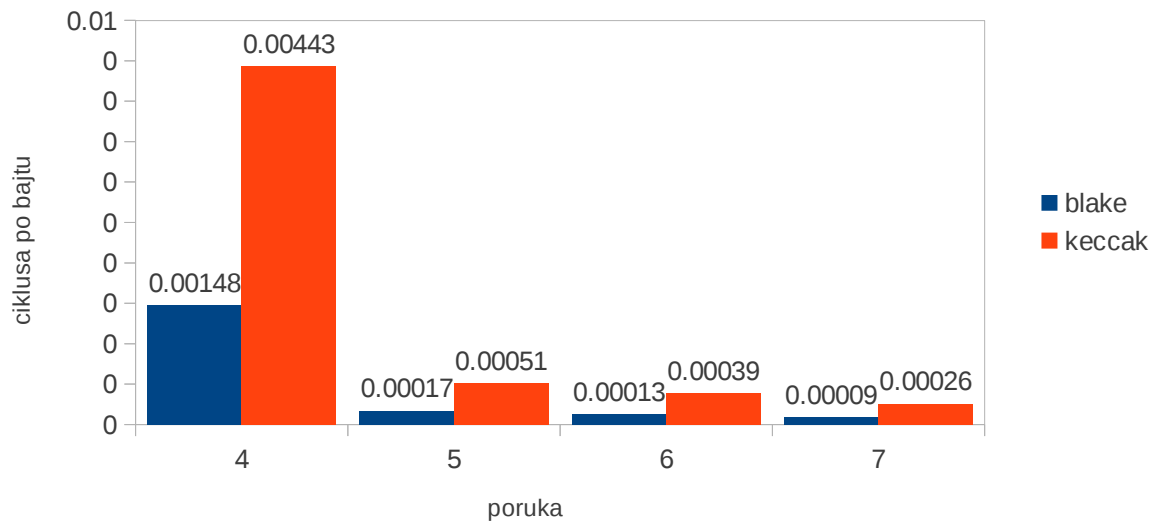
Veličina uzorka 101. 101 merenje po uzorku.

Ciklusa po bajtu



Ciklusa po bajtu za prve tri poruke

Ciklusa po bajtu



Ciklusa po bajtu za ostale poruke



Ciklusa po bajtu za sve poruke

Poruka (bajtova) / Hash	Blake	Keccak
1	204.3879084	394.7496953
65	3.9740471	3.9829588
163808	0.0006807	0.0018674
3122383	0.0000412	0.0000648
27189623	0.0000069	0.0000079
3596452	0.0000035	0.0000062
54533929	0.0000021	0.0000069

Standardna promena

Zaključak

Za sve poruke Blake je računao sumu dosta brže. Sa mnogo manje ciklusa po bajtu od Keccak-a. Promene su dosta male i ne utiču na rezultate, tako da možemo reći da je to zaista tako. Možemo reći da je Blake brži i za veće poruke ako pogledamo trend.

Gledajući Blake primećujemo da broj ciklusa po bajtu opada kako veličina poruke raste. To je zato što sumu veće poruke računamo za isto vreme. Predpostavljamo da bi se ovo održalo do nekoliko stotina megabajta. Generalno možemo primetiti ako poruka malo poraste ciklusi po bajtu naglo opadaju. Kada su poruke veće, recimo nekoliko gigabajta, dobijemo broj ciklusa tipičan za tu regiju, ako je sve ostalo isto. Isto je i za Keccak.

Zbog čega je Blake brži? Prvo možemo primetiti broj rundi po bloku. Blake ima četrnaest rundi po bloku, dok Keccak ima dvadeset i četiri. To je skoro dva puta više od Blake. Unutar runde Keccak izvodi trideset i devet petlji. To je ukupno devet stotina trideset i šest petlji. Blake to nema i u odnosu na njega predhodni broj je veći blizu šezdeset i sedam puta.

Tu je i optimizacija koju smo uspeali uraditi kod Blake-a. Uspeli smo izbaciti računanje pozicije u tabeli sa permutacijama. U rundi nemoramo računati koji red da koristimo. To smo izračunali unapred, samo možemo pročitati u bilo kojoj rundi.

Do sada možemo videti koji je hash bolji za koju primenu. Blake je bolji gde je brzina bitna. Za primer kod kontrole verzije softvera. Ako imamo tipičan projekat od nekoliko gigabajta trenutnu verziju možemo dobiti brzo. To je veoma bitno kad više ljudi radi na projektu. Potrebno im je znati gde se nalaze u svakom trenutku.

Sa druge strane Keccak je može biti korisniji gde je sigurnost bitna. Ima novu konstrukciju za koju još nisu poznati napadi. Primer bi bili digitalni potpisi. Hoćemo da ih je teško krivotvoriti.

Literatura

- [1] Aumasson, J.P. , Henzen L. , Meier W. , Phan R.C.W (2011) *Blake reference implementation.*
- [2] Bertoni G. , Daemen J. , Pitters M. , Aske G.V. (2011) *Keccak reference implementation.*