Универзитет у Београду
Математички факултет

Мастер рад

Криптографски хеш алгоритам SHA-3

Ана Станковић

ментор
др Миодраг Живковић

чланови комисије
др Александар Картељ
др Стефан Мишковић

18. септембар 2020.

Садржај

1 Увод

Налазимо се у ери у којој се свакодневница великог броја људи своди на коришћење интернета у многобројне сврхе, од обављања различитих послова (попут онлајн куповине, плаћања рачуна, учења...) до интерперсоналне комуникације. Упркос свим предностима које одликују интернет, постоје и мане његовог коришћења. Корисници остављају своје приватне информације које лако могу постати доступне неовлашћеним корисницима и бити злоупотребљене од стране истих. Кључну улогу у заштити поверљивости података има криптографија. С обзиром на широки асортиман активности које се могу обављати на мрежи, а које се не могу замислити без одговарајућих безбедносних мера, криптографија се све више изучава и напредује на многим пољима.

Једно од средстава којим се криптографија користи у омогућавању и олакшавању овакве свакодневнице је криптографска хеш функција о којој ће бити више речи у наставку. Представља математичку функцију која сажима произвољан број битова у мали број битова. Хеш функција обезбеђује интегритет поруке, тј. осигурава да порука која је примљена није успут измењена, за разлику од симетричне и асиметричне криптографије које штите тајност поруке.

Централна тема овог рада је криптографски хеш алгоритам SHA-3. Циљ рада је упознавање са основним концептима алгоритма, принципом његовог рада и његовим применама. У поглављу 2 описује се сунђер конструкције која лежи у основи алгоритма SHA-3. Поглавље 3 посвећено је самом алгоритму SHA-3, трансформацијама које користи и његовим инстанцама. Поглавље 4 описује неке примене алгоритма. Последње поглавље, поглавље 5, намењено је опису програмске реализације алгоритма SHA-3 и тражење колизије у смањеној верзији.

Неки од делова овог рада настали су на основу енглеске верзије текстова на Википедији[1][2][3]. Прилагођени преводи ових чланака постављени су на српску верзију Википедије[4][5][6].

2 Криптографске хеш функције

Један од важних алата за постизање безбедности информација у модерној криптографији су криптографске хеш функције. Значај су стекле проналаском криптографије са јавним кључем (асиметрична критографија) од стране Дифија (Bailey Whitfield Diffie) и Хелмана (Martin Edward Hellman) 1976. године. Било да се шифрује имејл порука, шаље порука на мобилном телефону, повезује се на HTTPS веб локацију или на удаљену машину преко IPSec или SSH, негде у оквиру поступка користи се хеш функција [1].

[1] https://en.wikipedia.org/wiki/SHA-3
[2] https://en.wikipedia.org/wiki/Sponge_function
[3] https://en.wikipedia.org/wiki/Authenticated_encryption
[4] https://sr.wikipedia.org/wiki/SHA-3
[5] https://sr.wikipedia.org/wiki/Сунђер_конструкција
[6] https://sr.wikipedia.org/wiki/Аутентификовано_шифровање

Хеш функција је математичка функција која се примењује на поруку променљиве дужине дајући излаз фиксне дужине који се назива хеш вредност. Важна је сигурност да примењена порука није измењена, тј. да јој је сачуван интегритет. Ово својство омогућава хеш функција упоређивањем добијеног хеша и израчунатог хеша. Хеш функција треба да:

- омогући брзо израчунавање вредности,
- буде једносмерна функција: тешко је одређивање x за задато y , такво да је $H(x) = y$,
- буде отпорна на колизију: тешко је наћи x' различито од x, такво да је $H(x') = H(x)$.

Без обзира на то која је хеш функција изабрана, колизије ће увек постојати. Треба направити хеш функцију тако да их је тешко наћи. Један од начина за проналажење колизије је рођендански напад. Ако је дато N порука и исто толико хеш вредности, може се размотрити укупно $N \cdot (N-1)/2 \approx N^2$ потенцијалних напада, узимајући у обзир сваки пар од две хеш вредности. Заснива се на рођенданском парадоксу или чињеници да ће група од 23 особе имати две особе са истим датумом рођења са вероватноћом од $1/2$. За дату n-битну хеш функцију h, груба сила тражи колизију израчунавањем функције h за случајне улазе све док се две исте хеш вредности не пронађу. Узимајући резултате рођенданског парадокса, можемо очекивати да ће се такве две вредности појавити након $(2^{n/2})$ $(\sqrt{2^n})$ итерација.

Хеш функције користе итеративну конструкцију, што подразумева дељење порука на блокове, а потом итеративну обраду сваког блока. Дизајн је приказан на слици 1 [14].

Две конструкције за итеративно хеширање које су најзаступљеније су Меркле-Дамгард конструкција и сунђер конструкција. Поред ових, постоји још конструкција у употреби, али су заступљене у мањој мери.

2.1 Меркле-Дамгард конструкција

Од настанка 1989. године па све до 2010. године, Меркле-Дамгард конструкција била је популарна конструкција већине тадашњих хеш функција. Творци ове конструкције су Ралф Меркле (Ralph Merkle) и Иван Дамгард (Ivan Damgård), независно један од другог.

Срж ове конструкције представља функција компресије $f : \{0,1\}^b \times \{0,1\}^n \to \{0,1\}^n$. На самом почетку, порука M се дели на блокове $M = M_1, M_2, ..., M_l$ једнаке дужине b. Уколико последњи блок није исте дужине као и претходни блокови, допуњује се. Један од примера како се обавља поступак допуњавања је следећи: на битове из последњег блока додаје се најпре јединица, затим нуле битова и на крају дужина поруке изражена у битовима. Допуњавање је потребно како би се обезбедило да две различите поруке дају другачије секвенце блокова, самим тим и различите хеш вредности [1]. Функција компресије извршава се n пута.

Слика 1: Итеративна конструкција хеш функције

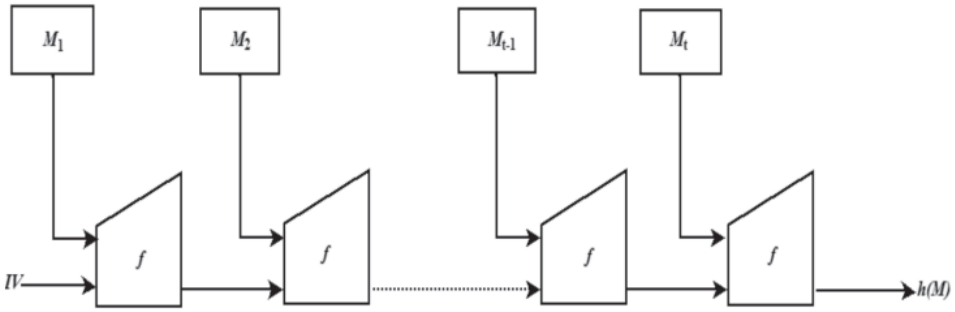
Приликом извршавања функције компресије, аргументи су $z_0 = IV$ и M_1. Хеш вредност је вредност која се добија после последњег извршавања функције компресије. Конструкција је приказана дијаграмом на слици 2, а псеудокод описан у алгоритму 1.

Меркле-Дамгард конструкција користи блоковску шифру као функцију компресије. Сигурност Меркле-Дамгард конструкција зависи од сигурности функције компресије. Дакле, да би се направила хеш функција отпорна на судare, довољно је да се направи функција компресије отпорна на колизије.

Неке од најпознатијих хеш функција које имају ову структуру су MD4, MD5, SHA-0, SHA-1, SHA-2. Међутим, након успешних напада на MD5, SHA-0, SHA-1 јавила се мотивација да се истражују нове хеш функције и нове конструкције, које ће имати боље карактеристике од Меркле-Дамгард конструкције и самим тим обезбедити већу сигурност против напада. Једна од таквих конструкција је сунђер конструкција.

Алгоритам 1 Функција компресије

1: $z_0 = IV$
2: **for** $i = 0$ to m
3: $\quad z_i = f(z_{i-1}, M_i)$
4: **return** $h(M) = z_M$



Слика 2: Меркле-Дамгард конструкција

2.2 Сунђер конструкција

Попут Меркле-Дамгард конструкције, сунђер конструкција користи се за итеративно хеширање, али за разлику од претходне, не користи функцију компресије и блоковску шифру већ функцију пермутације и потпуно је другачија. Настала је у циљу унапређења хеш функција.

Функција са оваквом структуром на излазу враћа бинарни низ произвољне дужине и назива се сунђер функција. Састоји се од три компоненте:

- меморије S, која садржи b битова,
- функције трансформације $f : \{0,1\}^b \to \{0,1\}^b$ која трансформише стање меморије (једна од $(2^b)!$ пермутација 2^b b-торки бита),
- функције допуњавања P.

Стање S састоји се од два дела: дела R, који означава део стања које се мења и чија је величина означена као брзина r (eng. *rate*) и дела C, који представља део стања који остаје непромењен при улазу/излазу са величином означеном као капацитет c (eng. *capacity*).

Сунђер функцију можемо означити као $SPONGE[f, pad, r]$, где је f функција трансформације, pad је проширење, а r је брзина.

Конструкција се састоји из две главне фазе: фазе упијања података (eng. *absorbing phase*) и фазе истискивања резултата (eng. *squeezing phase*). Аналогија са сунђером

је да се произвољан број битова на улазу упија у стање функције, а потом се произвољан број битова на излазу истискује из стања [13].

Пре прве фазе потребно је извршити припрему, која се огледа у томе да се стање S најпре иницијализује нулама, а затим се ниска допуњује помоћу функције допуњавања која додаје битове на крај ниске и на тај начин омогућава да се ниска подели на n блокова величине r битова.

У фази упијања података, за сваки блок B, величине r битова

- R се замењује са R XOR B,
- S се замењује са $f(S)$.

Када се сви блокови обраде, следи фаза истискивања резултата. Фаза истискивања резултата:

- Укључити део R стања S у излаз,
- заменити S са $f(S)$,
- Укључити део R стања S у излаз; ако у излазу недостаје мање од r битова, у излаз се укључује само део R,
- На крају скратити Z на d битова.

Битови из дела C не учествују у операцији XOR, нити се директно укључују у излаз, већ се мењају у зависности од функције трансформације f. Сигурност сунђер функције зависи од дужине њеног унутрашњег стања и дужине блокова и од хеш вредности. Ниво сигурности гарантован од стране сунђер функције је $c/2$. С обзиром да је за тражење колизије помоћу рођенданског напада потребно $2^{n/2}$ за n-битну функцију, сложеност је мања вредност између $O(2^{b/2})$, где је b дужина хеш вредности и $O(2^{c/2})$ [1].

Сунђер конструкција илустрована је дијаграмом на слици 3. У алгоритму 2 $Trunc_r(S)$ за позитиван r и стринг S означава ниску која је сажета на r бита, а $||$ представља конкатенацију ниски [13].

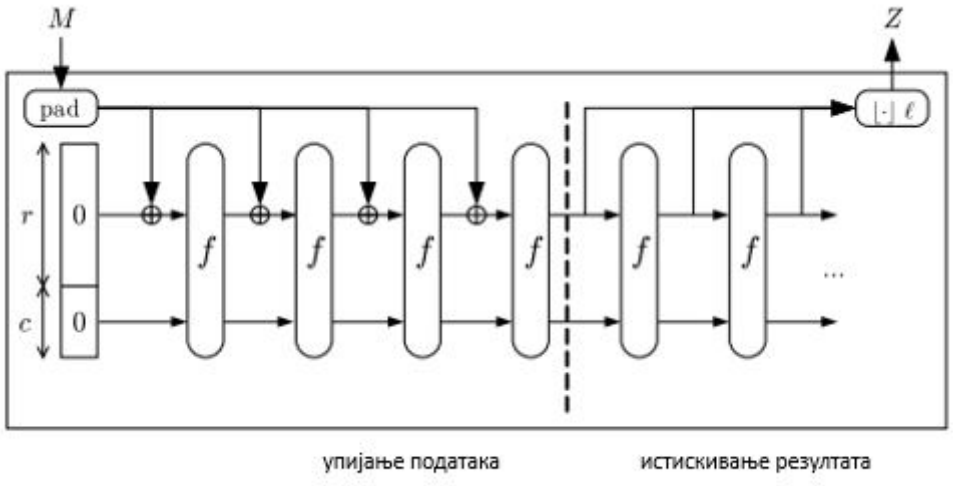
Једна од предности сунђер конструкције састоји се у томе што користи једноставну операцију XOR над битовима, уместо компликованих блоковских шифара које користи Меркле-Дамгард конструкција. Постоји још једна конструкција налик сунђер конструкцији, а назива се дуплексна конструкција или дуплексирање. Ова конструкција такође садржи функцију трансформације и параметар брзине r, али се разликује по томе што прихвата улазну ниску, а враћа излазну ниску која зависи од свих претходних улаза и чува резултат у објекту који се назива дуплекс објекат. Другим речима, наизменично се смењују фазе упијања и истискивања. Конструкција се може илустровати дијаграмом на слици 4.

Предности сунђер конструкције су следеће [2]:

- Једноставност – конструкција сунђера је врло једноставна,

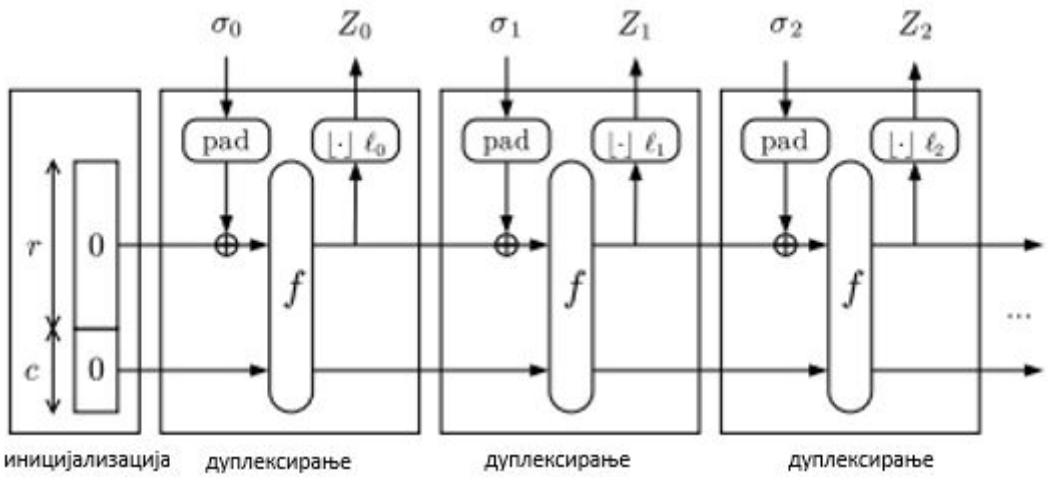
Алгоритам 2 Сунђер функција

Улаз: ниска N, ненегативан број d
Излаз: ниска Z тако да је $len(Z) = d$
$P = N||pad(r, len(N))$
$n = len(P)/r$
$c = b - r$
$P_0, P_1 \dots P_{n-1}$ је секвенца ниски, дужине r, тако да је $P_0||...||P_{n-1}$
$S = 0^b$
for $i = 0$ to $n - 1$
    $S = f(S \oplus (P_i||0^c))$
Z је празна ниска
$Z = Z||Trunc_r(S)$
while $d \leq len(Z)$
    $S = f(S)$
    $Z = Z||Trunc_r(S)$
return $Trunc_d(Z)$



Слика 3: Сунђер конструкција

- Произвољна излазна дужина – може да произведе резултате било које дужине, а самим тим се једна функција може користити за различите излазне дужине,
- Флексибилност – ниво сигурности се може повећати по цену брзине, подешавањем односа брзине и капацитета, користећи исту функцију трансформације,
- Функционалност – сунђер функција може се директно користити као MAC функција, проточна шифра, генератор псеудослучајних бројева.



Слика 4: Дуплекс конструкција

Сунђер конструкција због своје једноставности, нашла примену не само у имплементацији хеш функција, већ и код имплементације аутентикационих кодова порука, проточних шифара, генератора псеудослучајних бројева, аутентификованог шифровања.

3 Алгоритам SHA-3

Национални институт за стандарде и технологију (eng. *National Institute of Standards and Technology*, NIST) у Америци објавио је фамилију криптографских хеш функција, такозваних SHA (eng. Secure Hash Algorithm) 1993. године. Ови алгоритми коришћени су за стандард широм света.

Први чланови ове фамилије, SHA-0 и SHA-1 врло брзо су замењени стандардом SHA-2, након неколико успешно пронађених колизија. Бихам (Biham) је 2004. године приказао успешан напад на SHA-0, убрзо након тога Wang објављује први колизиони напад на SHA-1. С обзиром на то да SHA-2 користи исту конструкцију као и његови претходници (Меркле Дамгард конструкцију за коју се показала да није отпорна на нападе), јавила се потреба за алтернативом.

Иако успешан напад на SHA-2 још увек није откривен, NIST је одлучио да организује такмичење за нови стандард, SHA-3. Сврха SHA-3 је да може директно да замени SHA-2 у тренутним применама, када је то потребно, и повећа робусност NIST-овог скупа за хеширање.

Такмичење је почело 2. новембра 2007. године [5]. У првој рунди, 10. децембра 2008. године, изабран је 51 кандидат, од којих је отпало 5 кандидата 24. јула 2014. године, да би на крају, у последњој рунди остало њих 5: BLAKE, Grøstl, JH, Keccak

и Skein, децембра 2010. године. NIST је прогласио алгоритам Кечак (енг. *Keccak*) победником такмичења. Његови аутори су Гвидо Бертони (Guido Bertoni), Joan Дамен (Joan Daemen), Михаел Питерс (Michaël Peeters) и Жил Ван Аш (Gilles Van Assche). Базиран је на ранијим хеш функцијама, Panama и RadioGatun [9].

Године 2014. NIST објављује нацрт FIPS 202 „SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", који је годину дана касније, 5. августа 2015. године, одобрен и проглашен новим стандардом.

3.1 Опис алгоритма

Алгоритам Кечак потпуно се разликује од својих претходника, SHA-1 и SHA-2, првенствено због сунђер конструкције (видети тачку 2.2). Иако је конструкција такође итеративна, потпуно је другачија од Меркле-Дамгард конструкције, што је био и један од главних разлога зашто је алгоритам Кечак изабран за нови SHA-3 стандард.

Осим наведених предности сунђер конструкције (видети тачку 2.2), одлуци да алгоритам Кечак буде изабран за нови SHA-3 допринеле су и одличне перформансе хардвера, елегантнан и чист код, као и добре укупне перформансе.

С обзиром да је алгоритам добијен директном применом опште сунђер конструкције, ради на исти начин као што је описано у претходном поглављу. Пре прве фазе, врши се допуњавање ниске (погледати 3.3). У фази упијања, блокови поруке се XOR-ују са подскупом стања, а затим се трансформишу узајамно једнозначном функцијом трансформације f. С обзиром на то да је узајамно једнозначна, може се рећи да је пермутација. У фази истискивања, излазни блокови се читају из истог подскупа стања бита, при чему се после сваког ишчитавања примењује функција трансформације f.

У алгоритму Кечак, варирају три параметра:

- b – ширина функције,
- c – капацитет,
- n_r – број рунди у функцији.

За функцију трансформације f, која представља саму срж сунђер конструкције, изабрана је трансформација чији детаљан опис следи у тачки 3.2.

Алгоритам 3 представља псеудокод за $Keccak[r, c]$ сунђер функцију, са параметрима c (капацитет) и b (брзина), функцијом трансформације $Keccak - f[r + c]$ (где је $r + c = b$), улазном ниском M која је представљена као низ бајтова $Mbytes$, након чега следе крајњи битови $Mbits$. $Mbits$ омогућавају раздвајање домена између различитих инстанци. У фази допуњавања врши се допуњавање ниском облика 10*1 која се састоји од две јединице и нула између њих. Детаљније о томе може се прочитати у секцији 3.3.

Алгоритам 3 Keccak[r,c](Mbytes || Mbits)

Допуњавање
$P = Mbytes||Mbits||0x01||\ldots||0x00||0x80)$
Иницијализација
for $x = 0$ to 4
    for $y = 0$ to 4
        $S[x, y] = 0$
Фаза упијања
for сваки блок P_i из P
    for $x = 0$ to 4
        for $y = 0$ to 4
            if x+5*y $<$ r/w
                $S[x, y] = S[x, y]$ XOR $Pi[x + 5 * y]$
    $S = Keccak - f[r + c](S)$
Фаза истискивања
Z је празна ниса
while излазна величина није испуњена
    for $x = 0$ to 4
        for $y = 0$ to 4
            if x+5*y $<$ r/w
                $Z = Z||S[x, y]$
    $S = Keccak - f[r + c](S)$
return Z

3.2 Трансформација Keccak-f

Трансформцију Кечак означава се са Keccak-f[b], где b означава дужину излаза хеш функције у битима. За трансформацију Кечак карактеристична су два параметра: ширина трансформације (представља фиксну ширину ниске које се пермутује) и означава се са b и број рунди (број итерација) која се означава са n_r. Keccak-f$[b, n_r]$ је трансформација дефинисана за било која $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ и било који позитивни број n_r.

Постоје још два битна параметра који се користе у трансформацији и који су повезани са b: параметар w и параметар l, где је параметар $w = b/25$, а $l = \log_2(b/25)$.

Стање S се може представити као тродимензионална матрица битова величине $5 \times 5 \times w$ чији су индекси тројке (x, y, z) за које важи да је $0 \leq x < 5$, $0 \leq y < 5$, $0 \leq z < w$. Тада је елемент матрице $[x][y][z]$ бит улаза са редним бројем $(5x+y) \cdot w+z$. При томе су x, y, z редом индекси врсте, колоне и бита. Са индексима се рачуна по модулу 5 за прва два индекса, односно по модулу w за трећи индекс.

Стање S и његови делови, који представљају дво-димензионалне и једно-димензионалне низове, могу се видети на слици 5.
Једно-димензионални низови су:

- Врста (eng. *row*) - сет од 5 бита са константним y и z координатама,
- Колона (eng. *column*) - сет од 5 бита са константним x и z координатама,
- Трака (eng. *lane*) - сет од 5 бита са константним x и y координатама.

Дво-димензионални низови су:

- Плоча (eng. *sheet*) - сет од $5w$ бита са константним x координатама,
- Раван (eng. *plane*) - сет од $5w$ бита са константним y координатама,
- Парче (eng. *slice*) - сет од 25 бита са константним z координатама.

Рунда у трансформацији Кечак, са ознаком Rnd, састоји се од низа од пет трансформација које се називају пресликавања корака (eng. *step mappings*) [13], означених са θ, ρ, π, χ и ι у којима се нова матрица A' добија од старе матрице A:

$$A' = Rnd(A, i_r), \text{ где је } Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r) \quad ,$$

а i_r редни број рунде. Основна блок пермутација састоји се од n_r рунди. Број рунди зависи од параметра l:

$$n_r = 12 + 2l$$

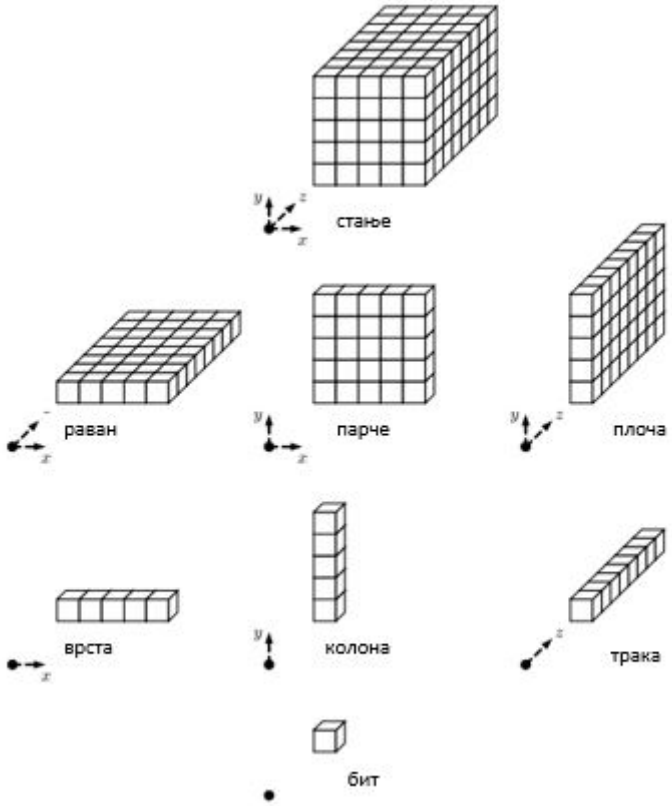
Функција трансформације користи операције XOR, AND и NOT. Алгоритам 4 представља псеудокод трансформација.

3.2.1 Корак theta

Нека је парност неког скупа бита једнака вредности XOR-а тих бита. Ако је вредност 0, онда је паран, а ако је 1, онда је непаран. Θ функција има за циљ

Алгоритам 4 Трансформације

1: **Theta**
2: **for** $x = 0$ to 4
3: $C[x] = A[x, 0]$
4: **for** $y = 1$ to 4
5: $C[x] = C[x]$ XOR $A[x, y]$
6: **for** $x = 0$ to 4
7: $D[x] = C[x - 1]$ XOR ROT $(C[x + 1], 1)$
8: **for** $y = 0$ to 4
9: $A'[x, y] = A[x, y]$ XOR $(D[x])$
10: **Rho**
11: $A'[0, 0] = A[0, 0]$
12: $(x, y) = (1, 0)$
13: **for** $t = 0$ to 23
14: $A'[x, y] =$ ROT $(A[x, y], (t + 1)(t + 2)/2)$
15: $(x, y) = (y, (3x + 3y))$
16: **Pi**
17: **for** $x = 0$ to 4
18: **for** $y = 0$ to 4
19: $A'[x, y, z] = A[y, (2x + 3y), z]$
20: **Chi**
21: **for** $x = 0$ to 4
22: **for** $y = 0$ to 4
23: $A'[x, y, z] = A[x, y, z]$ XOR $((NOTA[x + 1, y])$ AND $A[x + 2, y]$
24: **Iota**
25: $A'[0, 0] = A[0, 0]XORRC[i]$



Слика 5: Делови стања

да XOR-ује сваки бит стања са парношћу две колоне, при томе користећи само XOR операцију и операцију цикличног померања битова која представља ротацију операнта за један бит и означава се са $ROT([], 1)$. Помоћне променљиве $C[x]$ и $D[x]$ су једно-димензиони низови који садрже пет речи дужине w бита. Са индексима се рачуна по модулу 5, тако да $C[-1]$ означава $C[4]$. Слика 6 садржи шематску репрезентацију корака θ и приказује ротацију.

3.2.2 Корак rho

Корак ρ представља транслацију унутар сваке траке (речи), тј. врши се ротација бита тако што се на z координату додаје померај (eng. *offset*) који зависи од фиксираних x и y координата унутар траке. Као и у претходном кораку, рачуна се по модулу величине траке. Слика 7 приказује транслацију.

Табела 1 приказује резултат извршавања горе наведеног псеудокода, а у табели 2 могу се видети помераји транслације који су добијени формулом из претходног кода.

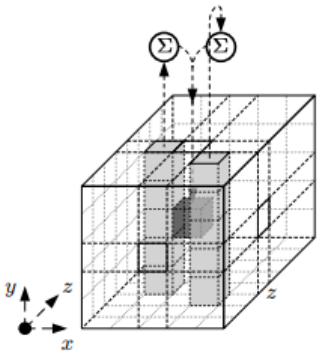
На слици 7 црним тачкама приказани су битови којима су z координате 0, а

t	$((t+1)(t+2)/2)$	(x, y)
0	1	(1,0)
1	3	(0,2)
2	6	(2,)1
3	10	(1,2)
4	15	(2,3)
5	21	(3,3)
6	28	(3,0)
7	36	(0,1)
8	45	(1,3)
9	55	(3,1)
10	2	(1,4)
11	14	(4,4)
12	27	(4,0)
13	41	(0,3)
14	56	(3,4)
15	8	(4,3)
16	25	(3,2)
17	43	(2,2)
18	62	(2,0)
19	18	(0,4)
20	39	(4,2)
21	61	(2,4)
22	20	(4,1)
23	44	(1,1)

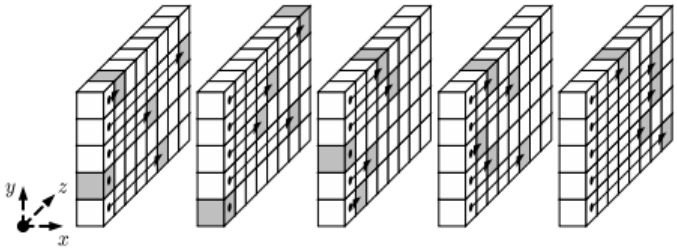
Табела 1:

	t	x = 1	x = 2	x = 3	x = 4
y = 4	18	21	61	56	14
y = 3	41	45	15	21	8
y = 2	3	10	43	25	39
y = 1	36	44	6	55	20
y = 0	0	1	62	28	27

Табела 2: Табела помераја



Слика 6: Корак θ примењен на појединачном биту



Слика 7: Корак ρ за случај w = 8

осенчаним коцкама су означене позиције тих битова након извршавања корака ρ, тј. колико су се померили унапред. Такође се и остали битови унутар траке померају за исти померај. На пример, ако погледамо у табели за $[0, 0]$, померај је такође 0, видећемо да се на слици црна тачка поклопила са осенчаном коцком (средина средње плоче), а да је за $[1, 0]$ померај 1 и да се осенчана коцка померила за једну позицију, што се може и видети у средини четврте плоче.

3.2.3 Корак pi

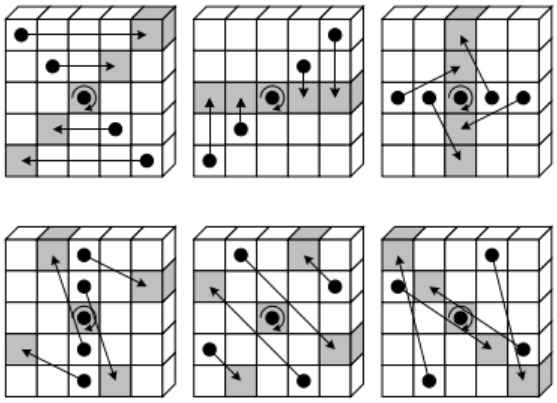
Корак π уско је повезан са кораком ρ јер пермутује ротиране траке (речи), тј. мења их.

Табела 3 приказује резултат извршавања корака pi.

Ако узмемо на пример $[1, 0]$, након примене формуле из кода, нова позиција постаје A'$[0, 2]$, што се може и видети на слици 8 која приказује пермутацију ротиране траке.

A[x,y]	A[y,(2*x + 3*y)]
A[0,0]	A[0,0]
A[0,1]	A[1,3]
A[0,2]	A[2,1]
A[0,3]	A[3,4]
A[0,4]	A[4,2]
A[1,0]	A[0,2]
A[1,1]	A[1,0]
A[1,2]	A[2,3]
A[1,3]	A[3,1]
A[1,4]	A[4,4]
A[2,0]	A[0,4]
A[2,1]	A[1,2]
A[2,2]	A[2,0]
A[2,3]	A[3,3]
A[2,4]	A[4,1]
A[3,0]	A[0,1]
A[3,1]	A[1,4]
A[3,2]	A[2,2]
A[3,3]	A[3,0]
A[3,4]	A[4,3]
A[4,0]	A[0,3]
A[4,1]	A[1,1]
A[4,2]	A[2,4]
A[4,3]	A[3,2]
A[4,4]	A[4,0]

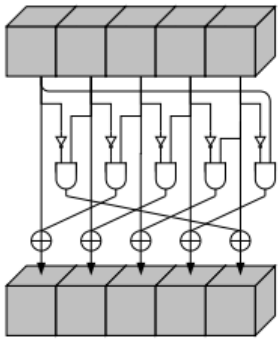
Табела 3: Пермутација



Слика 8: Корак π примењен на једну плочу

3.2.4 Корак chi

Наредни корак је једина нелинерана трансформација у оквиру SHA-3. Поред операције XOR, користе се и операције NOT и AND.



Слика 9: Корак χ

Циљ овог корака је да XOR-ује сваки бит са друга два бита у његовом реду. Приказ начина рада можемо видети на слици 9. Множе се негација суседног бита, што је представљено у коду као $(NOTA[x+1,y])$, и другог суседног бита $A[x+2,y]$, па се резултат XOR-ујемо са оригиналним битом.

3.2.5 Корак iota

У последњем, најједноставнијем кораку, додају се константе из табеле 4 на реч A[0,0]. Примењује се једноставна формула у којој i означава број тренутне рунде.

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

Табела 4: Константе рунде RC[i]

Константе из табеле су рачунате за $w = 64$, а за мање величине, једноставно се скраћују.

3.3 Допуњавање

Да би се подржала произвољна дужина ниске на улазу, потребно је извршити допуњавање ниске. Тај задатак спроводи *pad*, функција за допуњавање.

Споменуто је у поглављу 2.2 да је поруку произвољне дужине потребно продужити тако да нова дужина буде дељива са r. Функција Pad означава се са $pad10^*1$ и обухвата додавање јединице, па 0 или више нула (највише r-1 нула), па јединицу. Вишеструко допуњавање додаје најмање 2 бита, тако да ће се две јединице додати и у случају када је дужина поруке већ дељива са r. У том случају, додаје се наредни блок који почиње и који се завршава јединицом, између којих се налазе r-2 нуле. На овај начин се постиже да порука чија је дужина дељива са r, а која се завршава низом бита који изгледа као низ за допуњавање, не даје исту хеш вредност као порука са одстрањеним делом која личи на допуњавање. Прва додата јединица је неопходна да би се разликовале хеш вредности порука које се разликују само по броју нула на својим крајевима.

Алгоритам за допуњавање на улазу прима позитиван број r, ненегативан број d, дужина поруке, а излаз је ниска P за допуњавање, тако да је $d + len(P)$ дељиво са r. Ниска P се добија тако што се на јединицу надовежу j нула, где је j израчунато помоћу формуле $j = (-d - 2) \bmod r$, а затим се на самом крају дода јединица.

3.4 Стандардизоване инстанце

Фамилију SHA-3 чине четири криптографске хеш функције, SHA3-224, SHA3-256, SHA3-384, SHA3-512 и две „Прошириве излазне функције” (eng. *Extendable Output Functions*), SHAKE128 и SHAKE256.

Свака од функција SHA-3 заснива се на инстанци алгоритма Кечак. Број 3 у имену означава верзију сигурносних хеш алгоритама. Функције SHAKE (Secure Hash Algorithm KECCAK) су нова врста криптографских примитива. За разлику од претходних хеш функција, именоване су по својим отпорностима на колизиони напад [8]. Број на крају имена означава карактеристику хеш функције, тј. дужину израчунате хеш вредности. За функције SHA3-224, SHA3-256, SHA3-384, SHA3-512, вредности су редом 224, 256, 384, и 512, док број код проширивих хеш функција означава меру сигурности. SHAKE128(M, 256) може се користити као хеш функција са дужином хеша од 256 бита и укупном мером сигурности од 128 бита за поруку M, а SHAKE256(M, 256) са мером сигурности од 256 бита и дужином хеша од 256 бита .

Поред различитих дужина хеш вредности, алгоритми могу имати и различите величине стања: 25, 50, 100, 200, 400, 800 и 1600. У стандарду SHA-3 препоручена величина стања је 1600.

Пошто је величина стања збир брзине и капацитета, могу се такође бирати различите вредности за њих, али да у збиру буду једнаке величини стања. На брзину израчунавања хеш вредности дугачких порука највише утиче време израчунавања функције f и операције XOR стања S са блоковима од којих се састоји проширена порука. Различите инстанце одговарају различитим компромисима између брзине и сигурности, тако да се бирају у зависности од тога да ли је потребнија бржа или сигурнија функција. Већа брзина даје бржу функцију, али по цену мање сигурности (отпорности на колизије) и обрнуто, мање r је сигурније, али мање ефикасно јер се мање битова поруке XOR-ује са стањем пре сваке примене рачунски сложене функције f.

RawSHAKE128 и RawSHAKE256 су додатне инстанце, које још увек нису стандардизоване. Користе стабло за хеширање и паралелизам са циљем бржег израчунавања дужих порука.

Све инстанце додају неке битове поруци, чији крајњи десни део представља суфикс за раздвајање домена (eng. *domain separation suffix*), да би се онемогућило конструисање порука које дају исти хеш за различите инстанце хеш функције Кечак. Најпре се битови за раздвајање домена надовезују на ниску па се тек онда за ту надовезану ниску рачуна колико треба додати битова како би дужина ниске била дељива са r и онда се примењује фунција $pad10^*1$. За сада су дефинисани следећи суфикси:

...0 резервисан је за будуће примене, ...01 је SHA-3, а ...11 је RawSHAKE .

Функција $Keccak[c]$ дефинише четири SHA-3 хеш функције додавањем суфикса од два бита поруци M и навођењем дужине излаза:

$$\begin{aligned}
\text{SHA3-224}(M) &= \text{Keccak[448]}\ (M||01, 224),\\
\text{SHA3-256}(M) &= \text{Keccak[512]}(M||01, 256),\\
\text{SHA3-384}(M) &= \text{Keccak[768]}\ (M||01, 384),\\
\text{SHA3-512}(M) &= \text{Keccak[1024]}(M||01, 512).
\end{aligned}$$

Функција $Keccak[c]$ дефинише и две SHA-3 XOFs, SHAKE128 и SHAKE256, додавањем суфикса од четири бита поруци M, за било коју дужину излаза d:

$$\text{SHAKE128}(M, d) = \text{Keccak}[256]\ (M||1111, d),$$
$$\text{SHAKE256}(M, d) = \text{Keccak}[512]\ (M||1111, d).$$

У току такмичења било је дозвољено да такмичари мењају своје алгоритме и тако отклоне проблеме откривене у међувремену. Алгоритам Кечак имао је неколико измена [2] :

- број рунди је повећан са $12 + l$ на $12 + 2l$, ради конзервативније гаранције,
- допуњавање поруке је, уместо сложене схеме, замењено једноставним низом битова 10*1,
- параметар r је повећан до границе безбедности, уместо ранијег заокруживања на најближи степен двојке.

Поред ових основних функција, NIST је у децембру 2016. године објавио документ NIST SP.800-185 [8] са четири нове SHA-3 изведене функције. У документу су дефинисане две варијанте cSHAKE, cSHAKE128 и cSHAKE256, затим KMAC1281 и KMAC256, TupleHash128 и TupleHash256 и последња функција ParallelHash, са варијантама ParallelHash128 и ParallelHash256.
Ове функције представљају унапређене верзије функција које су дефинисане у FIPS 202. Све функције су изведене од основних функција SHA-3, имају две јачине безбедности 128 бита и 256 бита, подржавају излазе са променљивим дужинама и персонализоване ниске које дозвољавају кориснику да дефинише своју варијанту функције.

Прва функција, cSHAKE, поред два обавезна параметра (параметар X, који представља улазну ниску битова, и параметар L, број који представља тражену излазну дужину у битовима) поседује још два опциона параметра, N и S, који јој омогућују експлицитно одвајање домена. Параметар N је име функције као ниска битова и поставља се на вредност дефинисану од стране NIST-а и тиме се врши одвајање домена. Други опциони параметар S представља персонализовану ниску битова.

KMAC (KECCAK Message Authentication Code) је хеш функција са кључем заснована на алгоритму Кечак. Поред улазне ниске, тражене излазне дужине у битовима, персонализоване ниске битова, садржи и параметар K, кључ као ниску битова било које дужине, укључујући нулу. Поред тога може да има бесконачно дугачак излаз, па се може користити и као псеудослучајна функција (eng. *Pseudorandom function, PRF*).

Функција TupleHash је за хеширање торки улазних ниски, чији излаз зависи од садржаја и редоследа улазних ниски. На пример, TupleHash("abc","d") и TupleHash("ab","cd") дају излазе које не личе један на други.

ParallelHash функција дизајнирана је у циљу ефикаснијег хеширања коришћењем предности паралелизма доступним у модерним процесорима. Поред параметара који су наведени за претходне функције, садржи и параметар B који представља величину блока у бајтовима и може бити било који број такав да је $0 < B < 2^{2040}$. Функција дели улазну ниску на низ непрекидних, непреклопљених блокова дужине B и за сваки блок се засебно израчунава хеш вредност. Те вредности се на крају спајају и прослеђују алгоритму cSHAKE који генерише коначну хеш вредност функције.

3.5 Безбедност

Захтеве које је SHA-3 требало да испуни у вези са безбедношћу је да буде отпоран на колизије, напад са одређивањем инверзне слике (eng. *preimage*) и напад са одређивањем друге поруке са истим хешом (eng. *2nd preimage*). Поред испуњења захтева, функције SHA-3 отпорне су и на друге нападе, као што су напади са продужавањем поруке. Обе прошириве излазне функције SHA-3 отпорне су на ове нападе, SHAKE128 са јачином до 128 бита, а SHAKE256 са јачином до 256 бита. Уколико је параметар d који представља излазну величину довољно мали, јачина ће бити и мања од $d/2$ бита при колизионим нападима и d бита код напада са одређивањем друге поруке са истим хешом и напада са одређивањем инверзне слике (видети табелу 5).

Функција	Излазна величина	Јачина у битовима		
		Колизија	Preimage	2nd Preimage
SHA3-224(M)	224	112	224	224
SHA3-256(M)	256	128	256	256
SHA3-384(M)	384	192	384	384
SHA3-512(M)	512	256	512	512
SHAKE128(M, d)	d	min(d/2,128)	≥min(d/2,128)	min(d/2,128)
SHAKE256(M, d)	d	min(d/2,256)	≥min(d/2,256)	min(d/2,256)

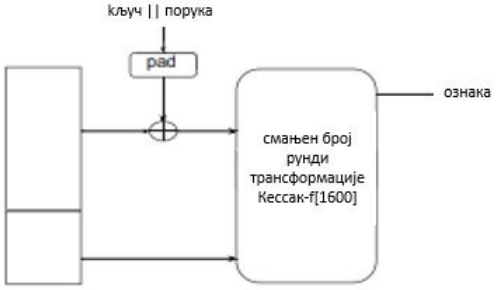
Табела 5: Безбедност функција SHA-3

4 Неке друге примене алгоритма Кечак

Кечак тим је предложио и додатне примене алгоритма које још увек нису стандардизоване. Поред своје примарне употребе као хеш функција без кључа, може се користити и у варијанти са кључем. У овој варијанти, сунђер функција може створити бесконачни низ бита, па се може користити као генератор псеудослучајних бројева, за проточне шифре. Такође, може се користити као саставни блок за аутентикациони код поруке (eng. *message authentication code*) и аутентификовано шифровање (eng. *authenticated encryption, AE*) [6].

4.1 Аутентикациони код поруке

Аутентичност поруке гарантује да подаци нису измењени од стране неовлашћеног субјекта, може се постићи функцијама које користе дељени тајни кључ и називају се аутентикациони кодови порука, MAC. MAC штити аутентичност поруке тако што креира ознаку сличну потпису. MAC који се заснива на алгоритму Кечак обезбеђује MAC функционалност тако што на самом почетку поруке додаје тајни кључ, након чега се примењује трансформација Keccak-f и израчунава ознаку (видети слику 10).



Слика 10: MAC заснован на Кечаку

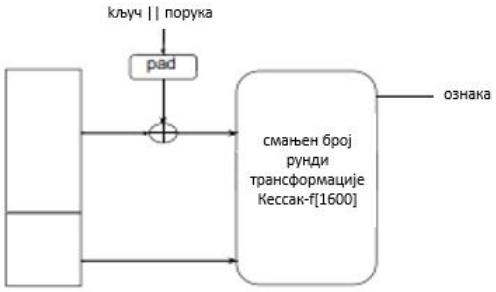
4.2 Проточна шифра

Проточна шифра се може формирати на основу алгоритма Кечак на следећи начин. Стање се иницијализује тајним кључем на који се надовезује иницијални вектор (IV) и након сваке примене трансформације Keccak-f добија се нови блок низа кључа за шифровање (eng. keystream), на који се примењује функција XOR са отвореним текстом дајући шифрат [6] (видети слику 11).

4.3 Паралелно извршавање

С обзиром на то да ће савремени процесори који подржавају паралелизам брже израчунати n пермутација истовремено него једну по једну, 2016. године Кечак тим је предложио две алтернативе (KangarooTwelve и MarsupilamiFourteen), које користе хеш стабло и тако омогућавају паралелно извршавање [4].

KangarooTwelve и MarsupilamiFourteen су проширive излазне функције, са функцијом трансформације Keccak-p$[1600, n_r]$, али са смањеним бројем рунди, 12 код KangarooTwelve, односно 14 код MarsupilamiFourteen. Брже су од стандардних FIPS 202 функција и за извршавање KangarooTwelve користи се само 0,55 машинских циклуса по бајту. Већа брзина је обезбеђена због паралелизма и због смањеног броја



Слика 11: Проточна шифра заснована на алгоритму Кечак

рунди. KangarooTwelve гарантује јачину од 128 бита, а MarsupilamiFourteen јачину од 256 бита.

4.4 Аутентификовано шифровање

Потреба да се истовремено обезбеди и поверљивост и аутентичност поруке у многим криптографским апликацијама је већа. Поверљивост података је гаранција да су подаци доступни само онима који су овлашћени за њих и постиже се шифровањем. Схеме аутентификованог шифровања (AE) су симетрични системи или системи са тајним кључем, помоћу којих се порука трансформише у шифрат на такав начин да се шифрату истовремено обезбеђује и поверљивост и аутентичност.

Опште решење аутентификованог шифровања може се конструисати комбиновањем алгоритма за шифровање и MAC под условом да је [10]:

- Алгоритам за шифровање отпоран на напад са изабраним отвореним текстом (eng. *chosen-plaintext attack*),
- Немогуће израчунати MAC на основу MAC произвољних других порука.

Белар (Bellare) и Нампремпр (Namprempre) су 2000. године анализирали три начина комбиновања шифровања и MAC примитива и показали да шифровање поруке, а затим примена MAC на шифрат (приступ Encrypt-then-MAC) обезбеђује сигурност против адаптивног напада са изабраним шифратом. Кац (Katz) и Јаунг (Yung) истраживали су појам под називом „шифровање отпорно на фалсификат“, за које су доказали да имплицира отпорност на нападе са изабраним шифратом [10].

Алгоритам AE има две операције: шифровање и дешифровање [11]. Операција шифровања на улазу прима три параметра:

- Тајни кључ који генерише случајна или псеудослучајна функција,
- Отворени текст, који садржи податке за шифровање и аутентикацију,

- Опционо заглавље отвореног текста које, иако се не шифрује, бива заштићено у погледу аутентичности.

Алгоритам на излазу даје шифрат и аутентикациони блок (аутентикациони код поруке).

Операција дешифровања прима четири параметра:

- Тајни кључ,
- Аутентикациони блок,
- Шифрат,
- Опционо заглавље, ако се користи током шифровања.

Алгоритам на излазу враћа отворени текст или грешку, ако аутентикациони блок не одговара шифрату или заглављу.

Варијанта аутентификованог шифровања која даје могућност да се провери интегритет придружених података који нису шифровани назива се аутентификовано шифровање са придруженим подацима (AEAD). Потребно је ове податке проверити иако остају отворени, како би уређаји за обраду могли правилно поступати са подацима. Схема AE може се модификовати додавањем MAC за аутентификацију отворених података. Ова схема, као и AEAD схема, користе нонс (eng. *nonce*), насумично генерисану вредност која се може користити само једном у криптографској комуникацији. Пошиљалац насумично генерише нонс и шаље примаоцу који га дешифрује договореним тајним кључем. Ово је неопходно да би се онемогућио напад са понављањем, у којем нападач, представљајући се као легитимни корисник, поруку поново шаље. Пошто је пошиљалац насумично генерисао нонс, тако спречава напад са понављањем јер нападач не може унапред да зна нонс који ће пошиљалац да изгенерише [15].

У оквиру стандарда ISO/IEC 19772:2009 предвиђено је шест различитих стандарда аутентификованог шифровања (OCB 2.0, Key Wrap, CCM, EAX, Encrypt-then-MAC (EtM), и GCM) [7].

CCM (Counter with Cipher Block Chain) је комбинација бројачког режима (eng. Counter Mode CTR) и режима уланчавања блока шифрата (eng. *Cipher Block Chaining CBC mode*). Бројачки режим користи се за енкрипцију и користи произвољан број који се мења за сваки блок поруке који се шифрује. Режим уланчавања блока шифрата користи се за обезбеђивање интегритета, конструише MAC помоћу блоковске шифре. Стандардизован је за блокове дужине 128 бита.

EAX је налик CCM, са једином разликом да користи OMAC (eng. *One-key CBC MAC*) за аутентификацију уместо CBC. OMAC дозвољава поруке произвољне дужине, док је MAC дефинисан само за поруке фиксне дужине.

GCM (eng. *Galois counter mode*) користи бројачки режим и GHASH за обезбеђивање интегритета. GHASH производи MAC користећи множење у $\mathrm{GF}(2^{128})$ који се дефинише као:

$$x^{128} + x^7 + x^2 + x + 1$$

OCB (енг. Offset Code Book) је стандард који омогућава различите блоковске шифре и дужине кључева. OCB 2.0 подржава придружене, некриптоване податке, за ралику од верзије OCB 1.0.

Key Wrap енкапсулира кључеве помоћу симетричне енкрипције. Један од познатијих алгоритама је AES Key Wrapping.

Постоје три приступа комбиновању алгоритма за шифровање и MAC: encrypt-and-MAC, MAC-then-encrypt и encrypt-then-MAC.

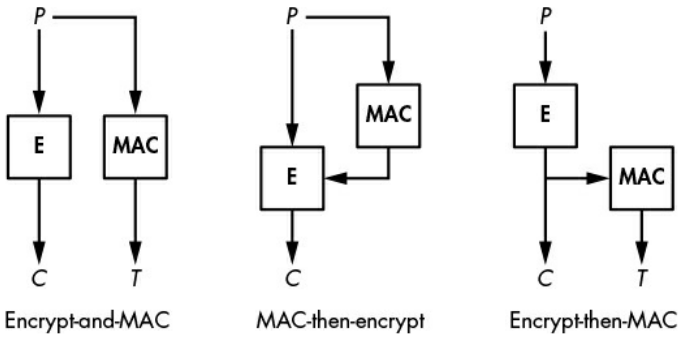
- SSL (или TLS), IPSec и SSH су популарни протоколи који примењују и симетричну аутентикацију и шифровање података, али сваки од ова три протокола то ради на другачији начин, тј. користи једну од наведених комбинација [10].

  IPsec користи Encrypt-then-MAC приступ. Отворени текст се прво шифрује, а затим се прави MAC на основу добијеног шифрата из резултата. Шифрат и његов MAC се шаљу заједно. Сматра се да је овај приступ најбезбеднији.

- SSL/TLS користи приступ који се назива MAC-then-encrypt (MtE). MAC се прави на основу отвореног текста, а затим се отворени текст и MAC заједно шифрују. Шаље се шифрат који садржи шифровани MAC.

- Encrypt-and-MAC је приступ који се користи у транспортном слоју SSH. MAC се прави на основу отвореног текста, а отворени текст је шифрован без MAC-а. Отворени текст MAC-а и шифрат се шаљу заједно.

Разлика се огледа у редоследу примене шифровања и MAC. Илустрација сва три приступа је приказана на слици 12 [1].



Слика 12: Комбинације аутентикације и шифровања података

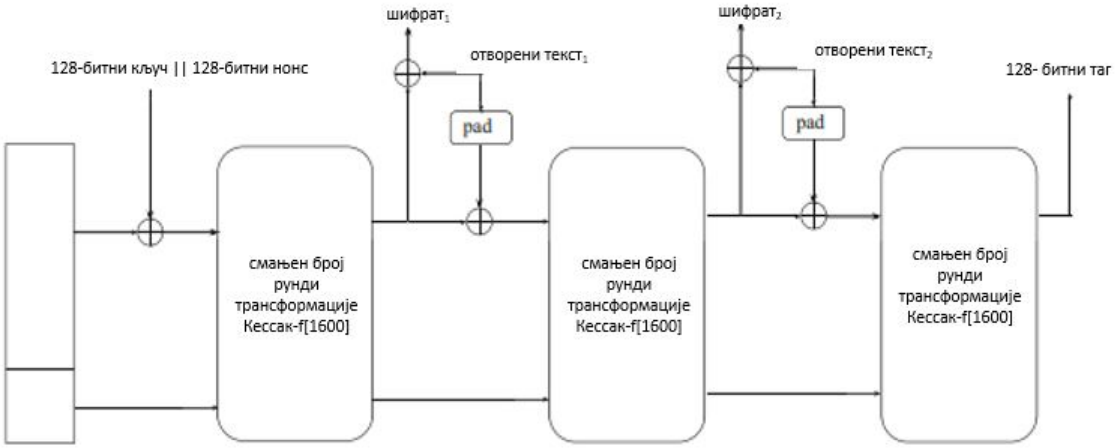
4.4.1 Аутентификовано шифровање помоћу алгоритма Кечак

Захваљујући сунђер конструкцији, односно дуплексној конструкцији, једна од примена Кечака је у изградњи аутентификованог шифровања. Кечак тим је обја-

вио неколико значајнијих схема аутентификованог шифровања које су базиране на трансформацији Кечак, са подесивим бројем рунда: Keyak, Ketje, Kravatte-SANE и Kravatte-SANSE.

С обзиром да то да AE комбинује шифровање порука и MAC, може се имплементирати помоћу дуплекс конструкције. Први корак је иницијализација која представља додавање кључа K и блока нонс N. Корисник мора поштовати захтеве за нонс које налажу спецификације ради обезбеђивања поверљивости. У спецификацији Keyak је наведено да се нонс не може поново користити.

Коришћењем дуплекс конструкције, AE захтева да се по једном позове трансформација f за сваки блок поруке. Укратко, улазни блокови дуплекса користе се за унос кључа и блокова поруке, а излазни блокови, осим последњег, користе се као низ кључева, док се последњи користи као MAC [3]. На слици 13 је схема Keyak за два блока података [6].



Слика 13: Схема Keyak

Између осталог, дуплекс конструкција подржава повезивање ниски које захтевају аутентикацију и шифровање и ниски које захтевају само аутентикацију па је самим тим згодно да се користи код аутентификованог шифровања са придруженим подацима. Тајни кључ K и блокови порука B_i обрађују се на следећи начин [12]:

- Кључ се упија (фаза упијања података),
- Блок придружених података A_i, дужине r битова, упија се (и остаје нешифрован). Затим се блок поруке B_i од r битова упија и шифрира са блоком од r битова који је био истискан из тренутног унутрашњег стања,

- Ознака T се враћа кад се заврши са процесирањем последњег блока и истискивањем блока од r битова.

5 Програмска реализација SHA-3 и експерименти

У овој секцији приказана је једноставна програмска реализација SHA-3 и на неколико примера је приказано да даје исте резултате као стандардна реализација из .NET библиотеке. Приказани су резултати експеримента тражење колизије са смањеном дужином излаза SHA-3 на нестандардне вредности 16, 24, 32, 40, 48.

5.1 Верификација једноставне реализације

Полазећи од једноставне реализације приказане у књизи[1], програм је имплементиран у програмском језику C#.

На улаз се уноси порука коју треба хеширати и величина хеша. Алгоритам је имплементиран у складу са стандардом, број рунди је 24, величина стања је 1600 бита, величина речи 64 бита. Најпре се врши иницијализација матрице која преставља стање и допуњавање поруке, а затим се позивају функције за упијање података и истискивање резултата. У наставку су дати псеудокодови ових функција. Комплетан код се налази у додатку А.

Функција за упијање података на улазу прима стање, допуњену поруку и величину блока, а враћа измењено стање.

```
Absorbing(state, messagePadded, blockSize)
{
    for (b = 0; b <  messagePadded.len / blockSize; b++)
        block = GetSubArray(messagePadded)
        for (i = 0; i < 5; i++)
          for (j = 0; j < 5; j++)
            if (i + j * 5 < (blockSize / 8)
                state[i,j] = state[i,j] ^(block[i+j*5])
            else break

        state = KeccakF(state)

    return state
}
```

За сваки блок који је XOR-ован са стањем, позива се функција трансформације KeccakF која на улазу прима стање у облику матрице. Састоји се од $12 + 2 * l$ рунди и свака рунда се састоји од пет корака. Унапред је дефинисан број рунди. По стандарду је 24, тако да је у програму исто наведена та вредност. У наставку је дат псеудокод .

```
    KeccakF( state)
       for (i = 0; i < numberOfRounds; i++)
          //Theta
          for (i = 0; i < 5; i++)
              C[i] = state[i, 0]^state[i, 1]^state[i, 2]^state[i, 3]^state[i, 4]
          for (i = 0; i < 5; i++)
              D[i] = C[(i + 4) % 5] ^ Rotate(C[(i + 1) % 5], 1, wordSizeInBits)
          for (i = 0; i < 5; i++)
              for (j = 0; j < 5; j++)
                  state[i, j] = state[i, j] ^ D[i];
          //Rho
           for (i = 0; i < 5; i++)
              for (j = 0; j < 5; j++)
                  state[i, j] = Rotate(state[i, j], offset[i, j], wordSizeInBits
)
          //Pi
          for (i = 0; i < 5; i++)
              for (j = 0; j < 5; j++)
                  B[j, (2 * i + 3 * j) % 5] = state[i, j]
          // Chi
          for (i = 0; i < 5; i++)
              for (j = 0; j < 5; j++)
                 state[i, j] = B[i, j]^((~B[(i + 1) % 5, j])&B[(i + 2) % 5, j])
          // Iota
          r = RC[i]
          state[0, 0] = state[0, 0] ^ r
       return state
```

Након фазе упијања података, следи фаза истискивања резултата, која враћа као излаз хеширану ниску жељене величине.

```
       Squeezing(state, blockSize, hashSize)
       {
           hashValue = []
           currentPositionInArray = 0
           for (i = 0; i < 5; i++)
               for (j = 0; j < 5; j++)
                  if (j+i*5 < (blockSize / wordSizeInBytes))
                        hashValue[currentPositionInArray] =(state[j,i])
                   currentPositionInArray+=1

           return hashValue
       }
```

У програму се налази функција која упоређује добијену вредност хеша са вредност хеша која је добијена помоћу готове функције из стандардне библиотеке SHA3.Net.

```
    CompareTwoAlgorithms(message) {
          s1 = Sha3.Sha3224()
          s2 = SHA3(message, 224)
          if (s1 == s2)
              Write("Oba algoritma daju isti rezultat")
          else
              Write("Algoritmi daju razlicite rezultate")
        }
```

У наставку су дати неки примери извршавања програма.

```
Za poruku "Prvi primer za hesiranje!" i vrednost 224 rezultat je:
ef8d01e9583d12f7fa8c85b0d01351add21016fa314cal5c94d64c64

Za poruku "Prvi primer za hesiranje!" i vrednost 256 rezultat je:
c909e1fe4f8c83e36b9cf2dfc43a74d748369269acb8cc8307c06148f44db2b8

Za poruku "Prvi primer za hesiranje!" i vrednost 384 rezultat je:
1c1e2ed9a0afeac649e55713db2fa067ef0b34931eaa387e80638886f7bfc41f5
4477cc205ea8747d8f951253cd8008ce

Za poruku "Prvi primer za hesiranje!" i vrednost 512 rezultat je:
7dbd406e9cea97171cd95073f04a43993c6c5d5dc3bcad063f47093b38a66c927
c5679eddf57372f0d2803b91d335ddf18904505aac1d9cca8b4c0b76b0ec31a

Za poruku "Danas je lep dan!" i vrednost 256 rezultat je:
a8c9b82720c3c1f35f24a30d45788b091e38081e93bc45f44b8c2c61e7a7cef

Za poruku "Danas je lepp dan!" i vrednost 256 rezultat je:
b1b68737f3d9cefab2e7ce2db57040c50fc8ecf72b9db89f157c846f9e2b4b1

Za poruku "Prvi primer za hesiranjesssssss222222sdsssaxccvvvvvvvvss
sswwwwwwwwwdsdsddsxxxxeeeqqqqaaaaaz" i vrednost 224 rezultat je:
dc99023e36879464ee054613d2f2197145db3b1bd2c2a1ba36dfab92

Poredjenje algoritma sa standardnom realizacijom iz .NET bibblioteke
Oba algoritma daju isti rezultat
```

Имплементација тражења колизија у програму заснива се на општем методу за тражење понављања у низу чији је наредни члан једнозначно одређен претходним чланом, а који се користи на пример у Полардовом ro методу за факторизацију целих бројева. Предност овог метода је мало заузеће меморије.

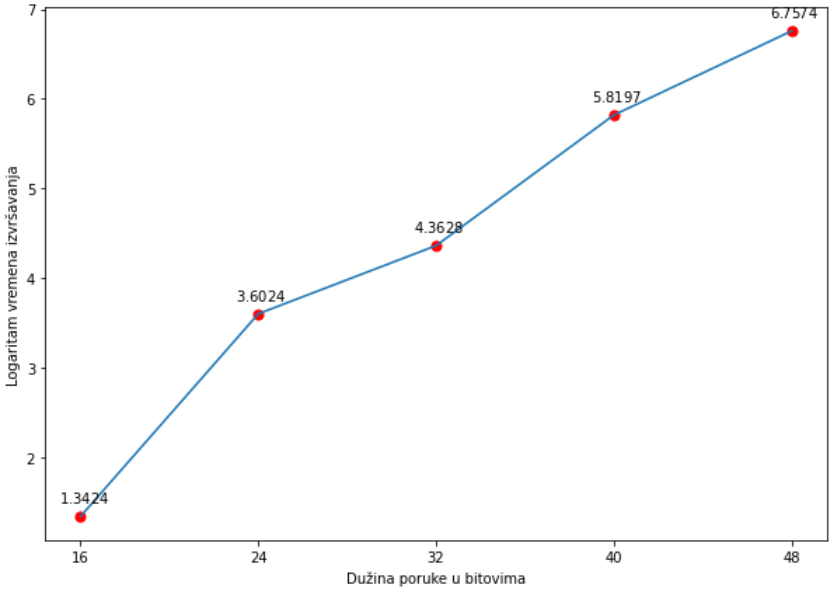
Дата је хеш функција са n битова (у програму су задате вредности 16, 24, 32, 40 и 48). Најпре се за поруку M_1 од n битова израчуна хеш вредност $M_1 = SHA3(M_1)$, а затим се израчуна $M_2 = SHA3(M_1)$. Следи итеративни процес у коме се израчунава $M_1 = SHA3(M_1)$ и $M_2 = SHA3(SHA3(M_2))$ који се прекида када ове две вредности буду једнаке. То значи да је пронађена колизија. У наставку је приказан псеудокод алгоритма за проналажење колизије и примери извршавања.

```
        public void FindCollision(string message, int hashSize){

            message1 = SHA3(message, hashSize)
            message2 = SHA3(message1, hashSize1)

            while (message1 != message2)
            {
                message1 = SHA3(message1, hashSize1)
                message2 = SHA3(message2, hashSize1)
                message2 = SHA3(message2, hashSize1)
            }

            message1 = message;
            while (message1 != message2)
            {
                message1 = SHA3(message1, hashSize1)
                message2 = SHA3(message2, hashSize1)

                if (message1 != message2)
                {
                  temp1 = message1;
                  temp2 = message2;
                }
                else if (SHA3(temp1, hashSize1) == SHA3(temp2,hashSize1))
                {
                  Write(temp1)
                  Write(temp2)
                  break
                }
            }
```

```
Prva niska je: ?} i njena hes vrednost je: 52f1
Druga niska je: êC i njena hes vrednost je: 52f1
Vreme nalazenje kolizije za poruku od 16 bita: 00:00:00:22
Prva niska je: d4 i njena hes vrednost je: 52973a
Druga niska je: «LV i njena hes vrednost je: 52973a
Vreme nalazenje kolizije za poruku od 24 bita: 00:00:04:03
Prva niska je: ãÝ¤? i njena hes vrednost je: 0e2f5653
Druga niska je: WºP? i njena hes vrednost je: 0e2f5653
Vreme nalazenje kolizije za poruku od 32 bita: 00:00:23:55
Prva niska je: !ÃJÂz i njena hes vrednost je: 7d1734aca3
Druga niska je: ?:ÅI? i njena hes vrednost je: 7d1734aca3
Vreme nalazenje kolizije za poruku od 40 bita: 00:11:02:98
Prva niska je: ?MsöAV i njena hes vrednost je: ab1aab15538e
Druga niska je: 4è???˜ i njena hes vrednost je: ab1aab15538e
Vreme nalazenje kolizije za poruku od 48 bita: 01:28:44:89
```

На дијаграму 14 приказана је зависност логаритма времена у милисекундама од

дужине хеша. Добија се приближно линеарна функција. Линеарни облик дијаграма одговара експоненцијалној зависности времена извршавања од дужине хеша.



Слика 14: Приказ зависности логаритма времена од дужине хеша

6 Закључак

У овом раду је описан криптографски хеш алгоритам SHA-3. Приказани су основни концепти и принцип рада алгоритма, као и неке од значајнијих примена алгоритма Кечак. Алгоритам је за потребе овог рада имплеметиран у оквиру окружења .NET. Приказан је његов рад на неколико примера, где се може видети да промена само једног слова или величине излаза битно мења хеш вредност. Вршено је упоређивање резултата са стандардном реализацијом из .NET библиотеке како би се тестирала тачност алгоритма. Урађено је тражење колизије са смањеном дужином излаза SHA-3 на нестандардне величине 16, 24, 32, 40 и 48 бита. Рад би могао да се даље развија са циљем убрзања израчунавања хеш вредности за дугачке поруке. Било би интересантно приказати неке од алгоритама, као што су KangarooTwelve и MarsupilamiFourteen, које омогућавају убрзање захваљујући паралелизацији.

Литература

[1] J. P. Aumasson. *Serious cryptography, A Practical Introduction to Modern Encryption.* William Pollock, 2018.

[2] G.M. Bertoni, Joan Daemen, and Michael Peeters. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 2009.

[3] G.M. Bertoni, Joan Daemen, Michael Peeters, and Gilles Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. volume 2011, pages 320–337, 2011.

[4] G.M. Bertoni, Joan Daemen, Michael Peeters, Gilles Assche, Ronny Keer, and Benoît Viguier. *KangarooTwelve: Fast Hashing Based on Keccak-p*, pages 400–418. 2018.

[5] R. Burr W. E. Turan M. Kelsey J. Paul S. Bassham L. Chang, S. Pelner. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition.* Commerce Department, National Institute of Standards and Technology (NIST), 2012.

[6] Itai Dinur, Pawel Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. In E Oswald and M Fischlin, editors, *Cube attacks and cube-attack-like cryptanalysis on the round-reduced Keccak sponge function. Advances in Cryptology - EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings, Part I [Lecture Notes in Computer Science, Volume 9056]*, pages 733–761. Springer, Germany, 2015.

[7] ISO Central Secretary. Information technology — security techniques — authenticated encryption. Standard ISO/IEC 19772:2009/COR 1:2014, International Organization for Standardization, Geneva, CH, 2014.

[8] R. Kelsey, J. Chang S. Perlner. *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.* Commerce Department, National Institute of Standards and Technology (NIST), 2016.

[9] Neha Kishore and Priya Raina. Parallel cryptographic hashing: Developments in the last 25 years. *Cryptologia*, 43(6):504–535, 2019.

[10] Hugo Krawczyk. In *The order of encryption and authentication for protecting communications (or: how Secure is SSL?).* Springer-Verlag, 2001.

[11] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, 2008.

[12] Pawel Morawiecki and Josef Pieprzyk. Parallel authenticated encryption with the duplex construction. Cryptology ePrint Archive, Report 2013/658, 2013.

[13] National Institute National Institute of Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions: FiPS PUB 202.* CreateSpace Independent Publishing Platform, 2015.

[14] Harshvardhan Tiwari. Merkle-damgård construction method and alternatives: A review. *Journal of Information and Organizational Sciences*, 41:283–304, 2017.

[15] Miles Tracy, Wayne Jansen, Karen Scarfone, and Theodore Winograd. Nist special publication 800-44 version 2, guidelines on securing public web servers. Technical report, 2007.

A Додатак

```
        private Byte numberOfRounds = 24; //po standardu je 24,12+12*l, gde je l=6
        private Byte wordSizeInBits = 64; // w = 2^l, po standardu je l=6
        private Byte wordSizeInBytes = 8;
        private UInt64[,] B = new UInt64[5, 5]; // pomocni niz

        // Tabela se racuna po formuli : ((t + 1)(t + 2)/2) % 64
        private Byte[,] offset = {
        {0, 36, 3, 41, 18},
        {1, 44, 10, 45, 2},
        {62, 6, 43, 15, 61},
        {28, 55, 25, 21, 56},
        {27, 20, 39, 8, 14}};

        private UInt64[] RC = {
        0x0000000000000001,
        0x0000000000008082,
        0x800000000000808A,
        0x8000000080008000,
        0x000000000000808B,
        0x0000000080000001,
        0x8000000080008081,
        0x8000000000008009,
        0x000000000000008A,
        0x0000000000000088,
        0x0000000080008009,
        0x000000008000000A,
        0x000000008000808B,
        0x800000000000008B,
        0x8000000000008089,
        0x8000000000008003,
        0x8000000000008002,
        0x8000000000000080,
        0x000000000000800A,
        0x800000008000000A,
        0x8000000080008081,
        0x8000000000008080,
        0x0000000080000001,
        0x8000000080008008};

        public string SHA3(string message, int hashSize)
        {
          //konvertujemo nisku u niz bajtova
            byte[] messageInBytes = StringToByteArray(message);

            int capacity = hashSize * 2; // formula za izacunavanje kacapiteta je
            // c = 2 * hashSize
            // stanje ima 1600 bita po stadardu, a rate se izracunava po formuli
            // 1600 = rate + capacity
            int rate = 1600 - capacity;
```

```
50              int blockSize = rate / 8; // velicina blokova u bajtima
52              // inicijalizujemo stanje S koje je predstavljeno kao
                // matrica S[i][j][k] bitova 5x5xw
54              UInt64[,] State = new UInt64[5, 5];
                for (Byte i = 0; i < 5; ++i)
56                  for (Byte j = 0; j < 5; ++j)
                        State[i, j] = 0;
58
                Byte[] messagePadded = new Byte[messageInBytes.Length + blockSize];
60
                Padding(messageInBytes, messagePadded, blockSize);
62              Absorbing(State, messagePadded, blockSize);
                return Squeezing(State, blockSize, hashSize);
64          }

66          public byte[] StringToByteArray(string str)
            {
68              return Encoding.UTF8.GetBytes(str);
            }
70
            //funkcija za dopunjavanje poruke kako bi se obezbedilo da bude
72          // deljiva sa r,  tj. blockSize
            private void Padding(Byte[] messageBytes, Byte[] messagePadd,int blockSize
    )
74          {
                int lastBlockSize = messageBytes.Length % blockSize;
76              int i;

78              // prekopirati nisku
                for (i = 0; i < messageBytes.Length; i++)
80              {
                    messagePadd[i] = messageBytes[i];
82              }

84              // ako je velicina poslednjeg bloka odgovarajuce velicine, dodaj 0x86
                if (lastBlockSize == 0)
86              {
                    messagePadd[i++] = 0x86;
88              }
                // ako je potrebno dopuniti blok do trazene velicine,
90              // dodaj najpre 0x06, zatim nule i na kraju 0x08
                // vrednosti su preuzete iz standarda FIPS 202
92              else
                {
94                  messagePadd[i++] = 0x06;

96                  while (blockSize - lastBlockSize - 2 > 0)
                    {
98                      messagePadd[i++] = 0x00;
                        lastBlockSize++;
100                 }
                    messagePadd[i++] = 0x80;
102             }
            }
104
            // funkcija koja deli niz bajtova na podnizove duzine length
106         public byte[] GetSubArray(byte[] bytes, int position, int length)
            {
108             var subArray = new byte[length];
                for (var i = 0; i < length; ++i)
110             {
```

```
                subArray[i] = bytes[i + position];
            }
            return subArray;
        }

    // funkcija za upijanje podataka
        private void Absorbing(UInt64[,] state,Byte[] messagePadded,int blockSize)
        {
            int numberOfBlocks = messagePadded.Length / blockSize;
            Byte[] block = new Byte[blockSize];
            int sizeStateInBytes = 8;

            // delimo poruku na blokove i svaki blok XOR-ujemo sa stanjem
            // zatim pozivamo funkciju transformacije Keccak-f
            for (int b = 0; b < numberOfBlocks; b++)
            {
                block = GetSubArray(messagePadded, b * blockSize, blockSize);
                int currentPosition;
                for (Byte i = 0; i < 5; i++)
                  for (Byte j = 0; j < 5; j++)
                  {
                    currentPosition = i + j * 5;

                    if (currentPosition < (blockSize / sizeStateInBytes))
                     {
                       currentPosition *= sizeStateInBytes;

                       for (Byte k = 0; k < sizeStateInBytes; k++)
                       {
                        state[i,j] ^=((UInt64)(block[currentPosition + k])<<(k*8))
                       }
                     }
                     else break;
                  }
                state = KeccakF(state);
            }
        }

        private UInt64[,] KeccakF(UInt64[,] state)
        {
            for (Byte i = 0; i < numberOfRounds; i++)
            {
                Theta(state);
                Rho(state);
                Pi(state);
                Chi(state);
                Iota(state, i);
            }

            return state;
        }

        // Theta korak
        private void Theta(UInt64[,] state)
        {
            UInt64[] C = new UInt64[5];
            UInt64[] D = new UInt64[5];

            for (Byte i = 0; i < 5; i++)
            {
                // C[i] je niz koji sadrzi 5 reci duzine wordSizeInBits
```

```
172                 C[i] = state[i, 0]^state[i, 1]^state[i, 2]^state[i, 3]^state[i,
    4];
            }
174         for (Byte i = 0; i < 5; i++)
            {
176             // D[i] je  niz koji sadrzi 5 reci duzine wordSizeInBits
                // Sa indeksima se racuna po modulu 5 pa je C[-1] isto sto i C[4]
178             // Rotate(C[], n, wordSizeInBits) oznacava rotaciju operanta za
                // n bitova po modulu wordSizeInBits
180             D[i] = C[(i + 4) % 5] ^ Rotate(C[(i + 1) % 5], 1, wordSizeInBits);

182         }
            for (Byte i = 0; i < 5; i++)
184             for (Byte j = 0; j < 5; j++)
                    state[i, j] = state[i, j] ^ D[i];//stane se XOR-uje sa nizom D
186     }

188     private UInt64 Rotate(UInt64 x, Byte n, Byte wordSize)
        {
190         UInt64 shiftLeft = x<<(n%wordSize); // siftovanje ulevo za n
            UInt64 shiftRight = x>>(wordSize-(n%wordSize));//siftovanje udesno za
    n
192         return shiftLeft | shiftRight;
        }
194

         // Rho korak
196     private void Rho(UInt64[,] state)
        {
198         for (Byte i = 0; i < 5; i++)
                for (Byte j = 0; j < 5; j++)
200             {
                //vrsi se rotacija bita tako sto se na z koordinatu dodaje pomeraj
202             // koji se cita iz tabele, ovde oznacene kao offset
                    state[i, j] = Rotate(state[i, j], offset[i, j], wordSizeInBits
    )
204             }

206     }

208     // Pi korak
        private void Pi(ulong[,] state)
210     {
            for (Byte i = 0; i < 5; i++)
212             for (Byte j = 0; j < 5; j++)
                {
214                 // vrsi se permutacija rotirane trake
                    B[j, (2 * i + 3 * j) % 5] = state[i, j];
216             }
        }
218

220     // Korak chi
        private void Chi(ulong[,] state)
222     {
            for (Byte i = 0; i < 5; i++)
224             for (Byte j = 0; j < 5; j++)
                {
226                  // XOR-ujemo svaki bit sa druga dva bita u njegovom redu
                     // sabiramo negaciju bita pored i bita dva mesta unapred
228                  state[i, j] = B[i, j]^((~B[(i + 1) % 5, j])&B[(i + 2) % 5, j])
                }
230     }
```

```
232         // Korak iota
            private void Iota(ulong[,] state, byte RC_i)
234         {
                var r = RC[RC_i];
236             // konstanta se cita iz tabele RC, a RC_i oznacava broj trenutne runde
                state[0, 0] = state[0, 0] ^ r; // dodajemo konstantu na rec state[0][0]
238         }

240         // funkcija koja ocekuje niz bajtova i koja vraca objekat tipa niske
            // koji sadrzi samo heksadecimalne brojeve
242         public string GetHexDigest(byte[] bytes)
            {
244           var str = string.Join(" ",bytes.Select(b=>string.Format("{0:x2}",b)));
              return str.Replace(" ", "");
246         }

248         // funkcija istiskivanja rezultata
            private string Squeezing(UInt64[,] state, int blockSize, int hashSize)
250         {
                // konvertujemo celo stanje u niz bajtova
252             Byte[] hashValue = new Byte[blockSize];
                int currentPositionInState = 0;
254             int currentPositionInArray = 0;
                for (Byte i = 0; i < 5; i++)
256                 for (Byte j = 0; j < 5; j++)
                    {
258                     currentPositionInState = j + i * 5;
                        if (currentPositionInState < (blockSize / wordSizeInBytes))
260                     {
                            for (int k = 0; k < 8; k++)
262                         {
                                var stemp = (state[j, i]) >> k * 8;
264                             hashValue[currentPositionInArray + k] = (Byte)stemp;
                            }
266                         currentPositionInArray += 8;
                        }
268                 }
                // pozivamo funkciju GetHexDigest za niz bajtova koja ih konvertuje u
270             //nisku sa heksadecimalnim brojevima i  izabranom velicinom hesa
                return GetHexDigest(hashValue.Take(hashSize / 8).ToArray());
272
            }
274         public void CompareTwoAlgorithms(string message)
            {
276           // algoritam iz SHA3.Net biblioteke
              // postoji i Sha3256(), Sha3384() i Sha3512()
278             var temp = Sha3.Sha3224();
                var s = temp.ComputeHash(Encoding.UTF8.GetBytes(message));
280             var s2 = string.Join(" ", s.Select(b => string.Format("{0:x2} ", b)));
                s2 = s2.Replace(" ", "");
282
                var s1 = SHA3(message, 224); // moj algoritam
284             if (s1 == s2)
                {
286                 Console.WriteLine("Oba algoritma daju isti rezultat\n");
                }
288             else
                {
290                 Console.WriteLine("Algoritmi daju razlicite rezultate\n");
                }
292         }
```

```
        public void FindCollision(string message1, int hashSize1){
            var message = String.Copy(message1);
            byte[] bMessage1;
            byte[] bMessage2;
            byte[] bMessage;
            var hes1 = "";
            var hes2 = "";
            bMessage1 = SHA3(message1, hashSize1).Take(hashSize1/8).ToArray();
            message1 = ByteToString(bMessage1);
            bMessage = bMessage1;
            bMessage2 = SHA3(message1, hashSize1).Take(hashSize1/8).ToArray();
            var message2 = ByteToString(bMessage2);

            Stopwatch stopwatch = new Stopwatch(); // za merenje vremena
            stopwatch.Start();
            while (!bMessage1.SequenceEqual(bMessage2))
            {
                bMessage1 = SHA3(message1, hashSize1).Take(hashSize1/8).ToArray();
                message1 = ByteToString(bMessage1);
                bMessage2 = SHA3(message2, hashSize1).Take(hashSize1/8).ToArray();
                message2 = ByteToString(bMessage2);
                bMessage2 = SHA3(message2, hashSize1).Take(hashSize1/8).ToArray();
                message2 = ByteToString(bMessage2);
            }
            message1 = String.Copy(message);
            bMessage1 = bMessage;
            var temp1 = "";
            var temp2 = "";
            while (!bMessage1.SequenceEqual(bMessage2))
            {
                bMessage1 = SHA3(message1, hashSize1).Take(hashSize1/8).ToArray();
                message1 = ByteToString(bMessage1);
                bMessage2 = SHA3(message2, hashSize1).Take(hashSize1/8).ToArray();
                message2 = ByteToString(bMessage2);

                if (!bMessage1.SequenceEqual(bMessage2))
                {
                  temp1 = message1;
                  temp2 = message2;
                }
                else if ((SHA3(temp1, hashSize1).Take(hashSize1/8).ToArray()).
                SequenceEqual(SHA3(temp2,hashSize1).Take(hashSize1/8).ToArray()))
                {
                  hes1=GetHexDig((SHA3(temp1,hashSize1).Take(hashSize1/8).ToArray
    ()

                  hes2=GetHexDig((SHA3(temp2,hashSize1).Take(hashSize1/8).ToArray
    ()
                  Console.WriteLine("Prva niska je: " + temp1 +
                   " i njena hes vrednost je:  "  + hes1);
                  Console.WriteLine("Druga niska je: " + temp2 +
                  " i njena hes vrednost je:   " + hes2);
                  break;
                }}
            stopwatch.Stop();
            TimeSpan ts = stopwatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes, ts.Seconds,
                ts.Milliseconds / 10);
            Console.WriteLine("Vreme nalazenje kolizije za poruku od {0} bita: "
             + elapsedTime, hashSize1); }
```