# Тестирање на криптографски хаш функции

Данко Илиќ

1. јули 2003

**Апстракт**

Се дефинираат криптографски хаш функции. Се наведуваат и објаснуваат нивните криптографски примени, и од тие примени се изведуваат критериуми за тестирање.

Потоа се покажува како практично може да се изведуваат тестирањата и се објаснува дизајнот и употребата на придружниот софтвер за тестирање.

На крај, се наведува користената литература и се даваат резултати од тестирањата изведени со софтверот врз некои познати и помалку познат хаш функции.

# Содржина

# Глава 1

# Дефиниции

Криптографските хаш функции се основен елемент во криптографските системи, подеднакво значајни како блок-кодовите, стрим-кодовите, криптирањето со јавен клуч и шемите за потпишување.

Во овој дел прво ќе дефинираме „обични“ хаш функции, кои не се ограничени само на криптографски примени, а потоа ќе ги претставиме својствата што е пожелно да ги има една криптографска хаш функција.

## 1.1 Хаш функции

**Дефиниција 1.1.1** ***Хаш функција, во потесна смисла****, е функција* $h : B_2^m \to B_2^n$ *која ги има следниве својства:*

1. ***компресија****:* $m, n \in \mathbb{N}$, $m > n > 0$*; т.е.* $h$ *пресликува влезен бит-стринг* $x \in B_2^m$ *со должина* $m$ *во излезен бит-стринг* $h(x) \in B_2^n$ *со должина* $n$*.*

2. ***пресметувачки едноставна****:* $\forall x$, $h(x)$ *лесно*[1] *се пресметува*

Излезот на хаш функција уште[2] се вика и *хаш вредност*, *хаш код*, или едноставно *хаш*.

Притоа, $|B_2| = 2$, а $B_2^k = \underbrace{B_2 \times B_2 \times \cdots \times B_2}_{k}$, т.е. $B_2^k$ е множество од сите бит-стрингови со должина $k$.

**Дефиниција 1.1.2** ***Хаш функција, во поширока смисла****, е функција* $h^\star : B_2^\star \to B_2^n$ *која ги има следниве својства:*

1. ***компресија****:* $n \in \mathbb{N}$, $n > 0$*; т.е.* $h^\star$ *пресликува влезен бит-стринг* $x \in B_2^\star$ *со* произволна должина $m$ *во излезен бит-стринг* $h^\star(x) \in B_2^n$ *со должина* $n$*.*

2. ***пресметувачки едноставна****:* $\forall x$, $h^\star(x)$ *лесно се пресметува*

[1] на пример, пресметувачката постапка има полиномијална комплексност

[2] hash, hash value, hash code, hash result, digest

Притоа, $B_2^\star = B_2 \cup B_2^2 \cup B_2^3 \cup B_2^4 \cup \cdots$, т.е. $B_2^\star$ е множество од сите бит-стрингови (со произволна должина).

Една хаш функција има задача да заштеди на простор, да може ефикасно да се пресметува, и – да имитира инјекција. Користејќи една позната аналогија, ако имаме повеќе гулаби отколку гулабови дупки, да направиме некако да изгледа како да има доволно гулабови дупки за сите гулаби.

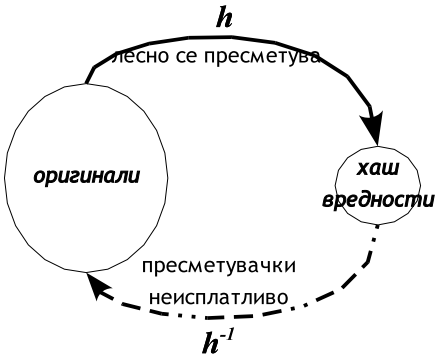
Една намена на „обичните“ хаш функции е да овозможат директен пристап до структури на податоци, при што треба да дојде до минимално пребарување на содржината, со скоро константна сложеност $O(const)$, за разлика од линеарната сложеност $O(n)$ при линеарно пребарување, или логоритамска $O(\log_2(n))$ при бинарно пребарување. Тие се користат во сите напредни системи за чување податоци, како големите пакети релациони бази на податоци, или интернет пребарувачите.

Друга намена на обичните хаш функции е детекцијата (и корекција) на грешки во разни комуникациски протоколи. Притоа, веројатноста да настане грешка при преносот, така што хаш вредностите на оригиналните и изменетите податоци ќе бидат исти, е многу мала[3].

Обичните хаш функции не се користат во криптографијата затоа што редовно имаат едноставна (линеарна) структура, т.е. затоа што се предвидливи. За криптографска употреба неопходно е тие да задоволуваат дополнителни својства.

## 1.2 Дополнителни својства

**Дефиниција 1.2.1** *За функција $h$ се вели дека е* ***еднонасочна***[4] *ако за „речиси секој“ однапред зададен излез, е „пресметувачки тешко“*[5] *да се најде влез што се хашира во тој излез.*




Сл. 1.1: еднонасочност

Значењето на терминот „пресметувачки тешко“, како и „пресметувачки едноставно“, треба да се разгледува во произволна референтна рамка, што

[3] затоа велиме дека „имитира инјекција“
[4] one-way, preimage resistant
[5] computationally infeasible

има практично значење во даден реален контекст. Таква рамка може да биде, како што често се зема, да се смета за пресметувачки тешка секоја постапка што има супер-полиномијална или експоненцијална комплексност, а за пресметувачки лесна постапка која има за комплексност полином од мал степен.

Терминот „речиси секој“ означува дека се остава можност за одредени специјални вредности да не важи својството, но тоа за најголемиот дел од вредностите треба да важи. Повторно, „најголемиот дел“ се одредува во однос на некоја избрана референтна рамка.

**Дефиниција 1.2.2** *За функција $h$ се вели дека е* ***слабо отпорна на колизии***[6] *ако е пресметувачки тешко, за дадено $x$, да се најде $x'$, $x \neq x'$, така што $h(x) = h(x')$.*

**Дефиниција 1.2.3** *За функција $h$ се вели дека е* ***силно отпорна на колизии***[7] *ако е пресметувачки тешко да се најдат $x$ и $x'$, $x \neq x'$, така што $h(x) = h(x')$. дадено $x$, да се најде $x'$ така што $h(x) = h(x')$.*

Јасно е дека ако една функција е силно отпорна, тогаш таа е и слабо отпорна на колизии, па често се користи и само терминот *отпорна на колизии.*

Функциите што се отпорни на колизии и се среќаваат во пракса *најчесто се и еднонасочни*, но строго формално тие не мора да бидат такви, што се гледа од следниов пример.

**Пример 1.2.1** *Нека $g$ е отпорна на колизии и дава излези со фиксна должина $n$. Ја дефинираме $h$ со:*

$$h(x) = \begin{cases} 1\|x & \text{, ако } x \text{ има должина } n \\ 0\|g(x) & \text{, инаку} \end{cases}$$

*Операторот $\|$ означува конкатенација на бит-стрингови.*

*Се гледа дека $h$ не е еднонасочна (за половина од сите хаш вредности), а е отпорна на колизии.*

## 1.3 Криптографски хаш функции

**Дефиниција 1.3.1** ***Еднонасочна хаш функција (OWHF***[8]***)*** *е хаш функција што е еднонасочна.*

**Дефиниција 1.3.2** ***Колизиски-резистентна хаш функциј (CRHF***[9]***)*** *е хаш функција која што е отпорна на колизии.*

Потребата од дефиницијата 1.3.1 е оттаму што во некои криптографски примени не се бара отпорност на колизии, туку само еднонасочност.

[6] weakly collision resistant, 2nd-preimage resitant
[7] strongly collision resistant, collision resitant
[8] one-way hash function
[9] collision resistant hash function

Сепак, сите познати и користени хаш функции се претпоставено CR, и под терминот „криптографска хаш функција“ обично се подразбира CRHF, иако може да се работи и само за OWHF.

Велиме „претпоставена“ затоа што не може да се докаже дека една хаш функција е безусловно сигурна. Имено, еден систем е безусловно сигурен ако не може да се пробие ниту во бесконечно време. Единствен познат криптографски примитив што е безусловно сигурен е one-time pad.

Но, од гледиште на пресметувачката сигурност[10], постојат повеќе хаш функции за кои може да се докаже дека се исто толку сигурни колку што е сигурен некој тежок проблем на којшто се базирани, како на пример проблемот на дискретни логаритми. За нив се вели дека се *докажливо сигурни*. Сепак, таквите хаш функции обично го имаат недостатокот што се *бавни*, и затоа во праксата се прибегнува кон користење специјално дизајнирани хаш функции, како MD5, SHA1, RipeMD, Tiger, кои што не се докажливо сигурни, но досега не се пробиени.

[10] computational security

Генералната конструкција на хаш функциите е *итерирачка*.

Мета-метод на Merkle за конструирање хаш функции во поширока смисла:

**влез:** CRHF $h : B_2^{n+r} \to B_2^n$ во потесна смисла

**излез:** CRHF $H : B_2^\star \to B_2^n$ во поширока смисла

1. Влез $x$ со бит-должина $b$ се разбива на блокови $x_1 x_2 \dots x_t$ од по $r$-бита; ако е потребно, последниот блок $x_t$ се дополнува со нулти битови.
2. Се дефинира екстра блок $x_{t+1}$ кој ја содржи десно-порамнетата бинарна репрезентација на бројот $b$. (претпоставуваме дека $b < 2^r$)
3. $H(x) = H_{t+1}$, каде

$$H_i = \begin{cases} \underbrace{00\cdots0}_{n} & , i = 0 \\ h(H_{i-1} \| x_i) & , 1 \leq i \leq t+1 \end{cases}$$

Ваквата конструкција може да се оправда со фактот дека е докажано дека, ако $h$ е CR, тогаш и $H$ е CR. $h$ често се вика и **функција за компресија**.

Значи, сигурноста на една хаш функција во поширока смисла потполно зависи од нејзината функција на компресија. Бидејќи наша цел е да ја тестираме сигурноста на CRHF, заради едноставност, ќе ја прифатиме терминологијата на специјализираната литература, и во критериумите за соодветност под **хаш функција** ќе подразбираме *функција на компресија на CRHF*.

# Глава 2

# Критериуми за соодветност

Критериумите[1] што треба да ги задоволуваат криптографските хаш функции доаѓаат од нивните примени. Затоа, прво ќе видиме кои се тие примени, а потоа ќе ги претставиме практично-проверливите[2] критериуми за нивна соодветност.

## 2.1 Видови примени на хаш функциите

**Податочен интегритет (data integrity)** Заедно со податоците се транспортира и нивната хаш вредност со цел да се детектира модификација на податоците (при трансфер).

**Електронски потписи (digital signatures)** Наместо да се потпишува одредена порака со произволна должина, така што заедно со пораката ќе се испраќа и потпис со иста должина како пораката, при дигиталното потпишување секогаш се користи хаш функција. Пораката се пропушта низ хаш функцијата, која дава фиксен излез, кој потоа се потпишува наместо самата порака. Ваквиот потпис е многу попрактичен за трансфер и, ако хаш функцијата е отпорна на колизии, тогаш тој е сигурен.

Доколку функцијата не е отпорна на колизии би било практично остварливо да се земе друга порака наместо оригиналната, и да се тврди дека всушност е потпишана таа, а не оригиналната.

Во оваа група примени спаѓаат и дигиталните отпечатоци[3] што служат за заштита на авторски права врз одредени дигитални материјали.

**Потврдување на знаење (confirmation of knowledge)** Како да се потврди сопственоста над одредени податоци, без да се открijат тие податоци самите? Доволно е да се споредат хаш вредностите; и да не се открие никаква информација за податоците. Пример за ваква употреба е верификацијата на лозинки за најавување на оперативните системи. На систем не

[1] certificational properties

[2] значи, не критериумите кои се само од теоретско значење, и за кои не постојат пресметувачки исплатливи постапки за одлучување

[3] digital fingerprinting

се чуваат лозинките, туку само нивните хашови. При најавување, се пресметува хаш од внесената лозинка и се споредува со хашот од вистинската лозинка. На тој начин супер-корисникот на системот, не може да ја дознае лозинката на корисникот, туку евентуално само да ја смени.

**Изведување клучеви (key derivation)** Понекогаш постои потреба за пресметување некоја нова криптографска тајна (клуч), врз основа на стара. Таков е случајот кај POS (point-of-sale) терминали. Тоа се посебни каси со кои се врши трансакција со смарт-картички. Бидејќи во такви ситуации редовно се бара сигурност на трансакцијата, хаш функциите се користат за генерирање нови сигурни криптографски клучеви врз основа на стари. Основата за оваа примена е тоа што процесорката моќ на смарт-картичката е ограничена, па својството на пресметувачка едноставност тука е клучно.

За таквите примени хаш функциите претставуваат најдобар и *најефикасен* генератор на случајни броеви.

**Генерирање на сигурни псевдо-случајни броеви (secure pseudo-random number generation)** Во сите области од применетата криптографија постои потреба од случајни броеви. На пример, за генерирање на прости броеви во RSA системот, или за генерирање сесијски клуч за симетрично криптирање. Специјализираните хаш функции од типот на MD5 или SHA-1 се одличен кандидат за таква улога, бидејќи се брзи, и еднонасочни. Всушност, многумина сметаат дека генераторот на случајни броеви е најкритичната алка во денешните применети криптографски системи. Исто така, хаш функции се користат и кога некој физички извор на вистински (не псевдо!) случајни броеви, треба дополнително да се ојакне.

## 2.2 Роденденски напад

Отпорноста на „роденденскиот напад“[4] е минимален критериум кој треба да го задоволува една хаш функција $h : X \to Z$ и кој дава една долна граница за нејзината сигурност, која зависи само од кардиналноста на множеството $Z$.

Името доаѓа од „роденденскиот парадокс“ кој тврди дека во група од 23 случајно избрани луѓе, со веројатност од барем $1/2$, барем двајца ќе имаат заеднички роденден.

Наједноставен начин да се најде колизија е да се земаат последователно различни вредности $x_1, x_2, \ldots \in X$ и да се пресметуваат вредностите $h(x_1), h(x_2), \ldots \in Z$. Бидејќи $h$ е сурјекција и $|X| > |Z|$, мора да се случи колизија, т.е. вистинит е исказот:

$$(\exists i)(\exists j)(x_i \neq x_j \wedge h(x_i) = h(x_j))$$

Веројатноста дека вклучително со $k$-тиот чекор од избирањето вредности нема да се случи колизија е:

$$\left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\cdots\left(1 - \frac{k-1}{n}\right) = \prod_{i=1}^{k-1}\left(1 - \frac{i}{n}\right)$$

[4] birthday attack

Познато е дека:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

За мали $x$, т.е големи $n$, може да апроксимираме: $1 - x \approx e^{-x}$, и тогаш:

$$\prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right) \approx \prod_{i=1}^{k-1} e^{-\frac{i}{n}} = e^{-\frac{k(k-1)}{n}}$$

Според тоа, веројатноста $\epsilon$ вклучително со $k$-тиот чекор да се случи барем една колизија е:

$$\epsilon \approx 1 - e^{-\frac{k(k-1)}{n}}$$

потоа

$$k^2 - k \approx n \cdot \ln\left(\frac{1}{1-\epsilon}\right)$$

Бидејќи ни треба само проценка, ќе ја поедноставиме равенката, занемарувајќи го мономот $-k$. Следува дека:

$$k \approx \sqrt{n \cdot \ln\left(\frac{1}{1-\epsilon}\right)}$$

За $\epsilon = 1/2$, проценката за вредноста на $k$ е:

$$k \approx 1.17\sqrt{n} \tag{2.1}$$

Тоа практично претставува математичко очекување[5] на бројот на чекори пред да се појави колизија. Што значи дека за наоѓање на колизија очекувано е да бидат потребни $O(\sqrt{n})$ време и меморија.

Ако сега се вратиме кај „роденденскиот парадокс“ и ставиме $n = 365$, добиваме дека $k \approx 22.3$, т.е. со веројатност од $1/2$ ќе се појави колизија, т.е. двајца луѓе ќе бидат родени на ист датум.

*

Добивме еден неопходен критериум што мора да го задоволува секоја хаш функција за да биде отпорна на колизии – да е пресметувачки тешко да се пребараат квадратен корен од бројот на сите можни хаш-вредности.

Контрастно, при напад со груба сила[6] во просек е потребно да се пребараат една половина од бројот на сите можни хаш вредности.

Според тоа, при дизајн на „сигурна“ хаш функција, мора да се води сметка множеството хаш-вредности да е доволно големо.

Денешните хаш функции типично имаат излез од 128 или 160 бита, а за повеќедеценијска сигурност се препорачува да се користат хаш функции со излез од 256 или 512 бита.

[5] за друг начин со којшто се стига до вредноста $\sqrt{n\pi/2}$, да се види [3]

[6] brute force attack

## 2.3 Барање колизии

Вашно е дека за обичниот вид на роденденски напад, освен временска, е потребна и $O(\sqrt{n})$ *мемориска* комплексност. Тоа претставува голем практичен проблем, бидејќи, како илистрација, за роденденски напад на хаш функција со 64-битен излез се потребни 4 гигабајти меморија, а за роденденски напад на 128-битна хаш функција се потребни $2^{61}$ бајти меморија[7]. Да не зборуваме дека меморијата е тесното грло на денешните (и идните) компјутери. Додека процесорската моќ постојано се зголемива, брзината на меморијата се зголемува незначително.

За среќа, постои начин да се постигне константна мемориска комплексност $O(const)$ при барањето колизии, со само неколкукратно зголемување на временската комплексност.

Ќе го користиме Pollard-rho[8] методот, кој се состои во тоа, наместо да ги земаме вредностите $x_1, x_2, \ldots$ според некое подредување на множеството на оригинали $X$ (на пример лексикографско), да работиме со

$$x_1, h(x_1), h(h(x_1)), \ldots$$

Се покажува дека ваквото движење низ просторот на оригинали не резултира со очекување за $k$ многу поинакво[9] од едноставното избирање на низата оригинали.

Само што сега може да се примени методот на Floyd[10] за наоѓање циклуси во периодични функции. Тој се состои во следното:

1. Фиксираме почетен оригинал $x_0$. $y \leftarrow x_0$, $z \leftarrow x_0$.
2. Пресметуваме: $y \leftarrow h(h(y))$, $z \leftarrow h(z)$.
3. Ако $y = z$, сме нашле колизија; ако не, одиме на чекор 2.

Докажано е дека двете низи ќе стигнат до иста точка, а за тоа ќе биде потребно отприлика 3 пати повеќе време, отколку во едноставната постапка, но со константна мемориска сложеност!

Овој начин на барање колизии е имплементиран во пакетот hashtest.

*

Се разбира, сите модерни хаш функции се направени да бидат отпорни на колизии, па е пресметувачки неисплатливо да се бараат целосни колизии. Но, при испитување на својствата на хаш функции корисно е да се откриваат и *делумни колизии*, т.е да се бараат хаш вредности коишто имаат определено Хамингово растојание[11]. Таква можност е имплементирана во пакетот hashtest.

[7] $\sqrt{2^{64}} = 2^{32}$, $\sqrt{2^{128}} = 2^{64}$

[8] оригинално создаден за проблемот на факторирање

[9] во $O()$ смисла

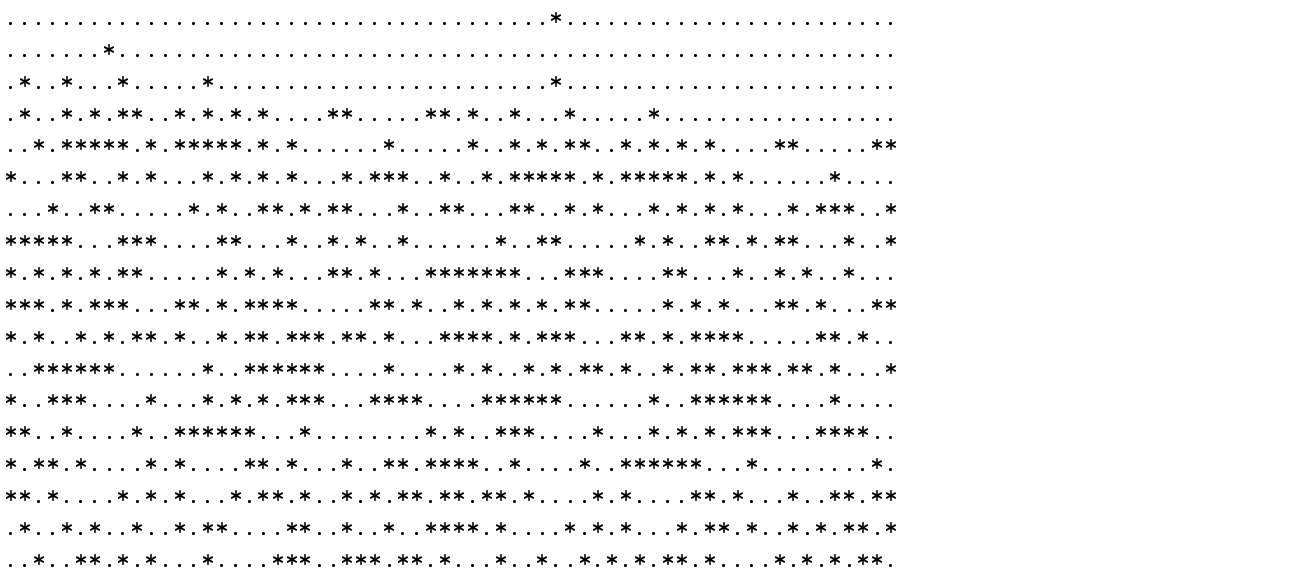
[10] за повеќе информации да се види [7], 3.1, пример 6

[11] Hamming distance; број на битови за кои се разликуваат два бит-стринга

## 2.4 Својство на лавина

Својството на „лавина“[12] претставува математичка апстракција што ја карактеризира (не)линеарноста помеѓу влезот и излезот на една хаш функција. И сликовито ќе се обидеме да опишеме зошто ова својство се вика токму така, на еден пример на хаш функција со 64-битен влез и излез.

Дадени се пресметувачките чекори во една итеративна хаш функција, т.е. се прикажани трансформациите на патот од оригинал до негова хаш вредност:



Во `hashtest` ова својство го мериме така што во секоја итерација од тестот се менува еден бит во влезот и се брои колку бита се промениле во излезот. Последнава вредност е случајна променлива за која што ги наоѓаме елементарните статистички показатели: просекот, дисперзијата и минималната и максималната вредност.

Во литературата се смета за добро својството промената на еден бит во влезот, во просек да менува половина битови во излезот. Јасно е дека е пожелно дисперзијата, т.е. расеаноста околу просекот, да биде што помала; во спротивно, во веројатносна смисла, се олеснува пронаоѓањето на (делумни) колизии.

## 2.5 Универзален статистички тест на Maurer

Не е возможно да се даде математички доказ дека еден генератор е навистина генератор на случајни броеви. Тоа е последица на проблемот што концептот за „случајност“[13] во математиката не е апсолутно дефиниран, туку од претпоставката дека е интуитивно јасен, врз него се градат (корисни)

[12] avalanche property

[13] убава разработка на филозофските и математичките концепти за „случајност“ е дадена во [7], 3.5. What is a random sequence?

теории како веројатноста, статистиката или теоријата на информации, кои потоа истиот го испитуваат!

[14]Ако земеме еден бинарен симетричен извор/генератор на битови, тој секоја конечна низа со должина $N$ ја генерира со иста веројатност $2^{-N}$. Според тоа, како би можело да се тврди дека одредена низа е „послучајна“ од друга низа?

Еден интересен пристап кон проблемот на дефинирање на „случајност“ за конечни низи е дело на Колмогоров. Тој неформално дефинира „случајност на низа“ како *должина на најкраткиот можен опис на правило за генерирање на таа низа.* Значи, една низа може да се смета за „случајна“ ако еден од најкратките описи е самата низа! Поформално, количеството на случајност, т.е. Колмогоров-комплексност, на една бинарна низа е дефинирано како должината на најкратката програма за дадена универзална Тјурингова машина, што ја генерира низата. Покажано е дека, во асимптотска смисла, низа која што е случајна според оваа дефиниција, ги задоволува сите статистички тестови за случајност што воопшто може да се конструираат.

Фундаментален проблем со Колмогоровата дефиниција за случајност, поврзан со фактот дека халтинг проблемот за Тјурингови машини е неодлучлив, е дека Колмогоров-комплексноста не е пресметлива (дури и со користење бесконечна пресметувачка моќ). Со други зборови, теоретски е невозможно, а не само пресметувачки неисплатливо, таа да се пресмета.

Но, и покрај сето ова, во праксата се користат статистички тестови за случајност. Причината за ова е што во многу случаи е разумно да се претпостави дека ако еден генератор е *дефектен*, тој се однесува[15] како бинарен извор без меморија или ергодичен стационарен извор со конечна меморија и непознати веројатности на премин меѓу состојбите.

Попознати тестови се: тестот на фреквенции, серискиот тест, покер тестот, автокорелационите тестовите и тестот со случајно талкање[16].

*

Основната идеја зад Мауреровиот „универзален“ статистички тест е дека една низа којашто е случајна не би требало да може значително да се компресира (без губиток на информации). Но, наместо да се обиде да ја компресира низата, Мауреровиот тест пресметува една статистика којашто е во врска со компресирањето, т.е. со ентропијата.

Тој е наречен „универзален“ затоа што, покрај сите дефекти детектирани од спомнатите статистички тестови, тој детектира и цела класа други дефекти.

Погодност на тестот е што е брз – има константна мемориска и линеарна (со должината на низата) временска сложеност.

Обично е недостаток на Мауреровиот тест што бара релативно долги низи ($> 10010 \cdot 2^L$), но тоа во случајот на тестирање хаш функции не е

[14]уводниот дел се состои од парафраза на воведот на трудот на Маурер, [2]

[15]Маурер дури (неформално) тврди дека секој дефект што би можел да се појави при практична конструкција на еден генератор, може да се опише како ергодички стационарен извор со конечна меморија, т.е. дека неговиот тест може да го „фати“ секој дефект во генератор што може реално да се конструира

[16]run test

недостаток, зашто не е наша намера да тестираме на база na малку итерации.

*

Еден Мауреров тест $T_U$ е определен од 3 параметри: $L$, $Q$ и $K$. Генерираната излезна низа од битови, $s^N$, се дели на соседни блокови од по $L$ бита. Должината на оваа низа е $N = (Q + K)L$, каде $Q$ е број на чекори за иницијализација, а $K$ е број на чекори на (остатокот од) тестот.

Нека $b_n(s^N)$ е $n$-тиот блок од низата $s^N$, $1 \leq n \leq Q + K$. За $Q + 1 \leq n \leq Q + K$, се наоѓа величината $A_n(s^N)$:

$$A_n(s^N) = \begin{cases} n & ,(\nexists i)(1 \leq i \leq n)(b_n(s^N) = b_{n-i}(s^N)) \\ min\{i : i \geq 1, b_n(s^N) = b_{n-i}(s^N)\} & ,\text{инаку} \end{cases}$$

која што ја претставува оддалеченоста на $n$-тиот блок од неговото последно појавување во низата $s^N$. Се дефинира статистиката

$$X_u(s^N) = \frac{1}{K} \sum_{n=Q+1}^{Q+K} \log_2 A_n(s^N)$$

Кога $s^N$ е случајна низа (пишуваме $s^N = R^N$), тогаш:

$$X_u \sim \mathcal{N}(\mu, \sigma^2) \tag{2.2}$$

каде $\mu$ и $\sigma^2$ зависат од должината на блоковите, $L$, и се дадени во следнава табела[17]:

| $L$ | $\mu$ | $\sigma^2$ |
|---|---|---|
| 1 | 0.7326495 | 0.690 |
| 8 | 7.1836656 | 3.238 |
| 16 | 15.167379 | 3.421 |

За да се постигне максимално ниво на детекција на дефекти, а и за да се минимизираат нумеричките грешки, нашата имплементација на Мауреровиот тест е за $L = 16$.

При имплементацијата во меморија не се чува целата низа $s^N$, туку се чуваат само индексите на последните појавувања на блоковите $b_n(s^N)$, што е доволно за да се пресметуваат вредностите $A_n(s^N)$. Тоа се прави со табела чија големина е еднаква на бројот на различни блокови, а тоа е $2^L = 2^{16} = 65536$, и, од таа гледна точка, мемориската комплексност е константна.

Всушност, првите $Q$ иницијализациски чекори служат за создавање на табелата, а долната граница на вредноста на $Q$ е условена од веројатност секој можен блок да се појави барем еднаш во првите $Q$ блокови од низата.

За практична примена препорачано е $Q > 10 \cdot 2^L (= 655360)$, а $K > 100 * Q$. Колку што е поголема вредноста на $K$, толку тестот е поточен, барем од теориска гледна точка, зашто знаеме дека нумеричките грешки се натрупуваат при пресметување на компјутер со фиксна прецизност.

[17] Овие вредности се експериментално пресметани. За повеќе информации да се види оригиналниот труд на Маурер.

Иако се работи за најмодерниот познат тест за испитување на случајност на една низа битови, и е генерализација на познатите тестови за случајност, важно е да се потенцира дека тестот детектира дефекти на ергодичен стационарен извор со меморија од $M \leq L$ бита. Пронаоѓањето на дефекти на модели кои имаат повеќе од $L$ бита меморија бара дополнителни[18] испитувања и симулации за да се утврди математичкото очекување и дисперзијата на нормалната распределба за таква вредност на параметарот $L$.

## 2.6 Статистичка Möbius анализа

Во минатогодишниот труд на група француски истражувачи, насловен „Ново статистичко тестирање на симетрични кодови и хаш функции“[1] се дава една нова класа на тестови, со чија помош статистички се најдени структурни нестабилности[19] во DES, AES и Lili-128. Во случајот на Lili-128 импликациите на тестот се потврдени со комплетно нова, детерминистичка и оперативна криптоанализа на овој код.

Познато е дека хаш функциите мора да се однесуваат така што да бидат генератори на псевдо-случајни броеви/битови. Со претходниот Мауреров статистички тест за униформност можат да се дадат некои карактеризации за тоа дали одредена низа битови е „случајна“. Новите тестови, што тука се претставуваат, се изградени на основа на теоријата на случајни Булови функции и нивниот развој во алгебарски нормална форма (ANF) со помош на Möbius-ова трансформација.

Една функција $f : B_2^n \to B_2$ претставува Булова функција. Бројот на такви функции е $2^{2^n}$.

**Дефиниција 2.6.1** ***Алгебарска нормална форма (ANF)*** *на $f$ е полином зададен со:*

$$f(x) = \bigoplus_{u \in B_2^n} a_u x^u$$

*$a_u \in B_2$, $u, x \in B_2^n$ и $x^u = \prod_{i=1}^n x_i^{u_i}$, каде $a_u$ се зададени со Мебиусовата трансформација на $f$:*

$$a_u = \bigoplus_{x \preceq u} f(x) \tag{2.3}$$

*каде $\preceq$ е релација во $B_2^n$ дефинирана со:*

$$\alpha \preceq \beta \iff (\forall i \in \{1, 2, \ldots, n\})(\alpha_i \leq \beta_i)$$

**Дефиниција 2.6.2** *Моном $a_u x^u$ од ANF има* ***степен k*** *акко $a_u = 1$ и $wt(u) = k$, каде wt(u) е Хамингова тежина[20] на $u \in B_2^n$.*

**Теорема 2.6.1** *Бројот на мономи со степен $k$, $n_k$, во алгебарската нормална форма на една Булова функција $f : B_2^n \to B_2$, има распределба $\mathcal{N}(\frac{1}{2}\binom{n}{k}, \frac{1}{4}\binom{n}{k})$.*

[18] за што, секако, треба да се консултира изворниот текст [2]
[19] biases
[20] бројот на ненулти битови во една низа од битови

**Последица 2.6.1** *Една Булова функција што се користи за криптографски примени, и што претставува најдобар баланс на криптографски својства, мора да има што е можно поголем степен.*

Со други зборови, ако во функција се воведуваат комбинаторни структури, тогаш нејзината случајност се намалува.

Хаш вредноста на една хаш функција $h : B_2^n \to B_2^m$ со влез $x = (x_0, x_1, \ldots, x_{n-1})$ може да се претстави на следниов начин:

$$h(x) = (h_0(x), h_1(x), \ldots, h_{m-1}(x))$$

За да тестираме една хаш функција треба да тестираме $m$ Булови функции. Притоа, ќе претпоставиме дека овие булови функции (т.е. битови) се статистички независни. Овој предуслов е токму цел на претходно дадените општи статистички тестирања, како Мауреровото тестирање.

Од практични причини[21] не е можно целосно пресметување на излезните ANF, бидејќи таков напор има пресметувачки и мемориски експоненцијална сложеност. Во нашите тестови ќе се фокусираме на мономи со степен најмногу 3, и ќе пресметуваме само парцијални ANF (чиишто коефициенти ефективно се пресметуваат до степен 3).

При реализацијата се служиме со развиената форма на формулата 2.3:

$$a_u = f(\overline{0}) \oplus \bigoplus_{j=1}^{k} f(e_{i_j}) \oplus \left( \bigoplus_{l=1}^{k} \bigoplus_{\substack{j=1 \\ j \neq l}}^{k} f(e_{i_j} \oplus e_{i_l}) \right) \oplus \cdots \oplus f\left( \bigoplus_{j=1}^{k} e_{i_j} \right)$$

каде $\overline{0} = \underbrace{0, 0, \ldots, 0}_{n}$, $\{i_1, i_2, \ldots, i_k\}$ е множеството индекси на единици во бинарната репрезентација на $u$, а $e_i \in B_2^n$ е битстринг со единствен бит 1 на $i$-тата позиција.

Нека $H_0^d$ е статистичката хипотеза дека бројот $\hat{n}_d$ на мономи со степен од точно d, е распределен според теоремата 2.6.1.

Ја изведуваме следнава постапка:

1. Ги пресметуваме $\nu$-те случајни променливи $n_d^i$ и $\hat{n}_d^i$.
2. Ја пресметуваме статистиката $D^2$ зададена со

$$D^2 = \sum_{i=1}^{\nu} \frac{(n_d^i - \hat{n}_d^i)^2}{n_d^i}$$

3. Фиксираме ниво на значајност $\alpha$ и праг[22] $x_\alpha$ (кој се пресметува од инверзната (кумулативна) функција на распределба на $\chi^2$-распределбата), така што кога случајна променлива X има $\chi^2$-распределба со $\nu$ степени на слобода, $P\{X > x_\alpha\} = \alpha$

[21] типична хаш функција има влез од 512 бита: за степен 1 ќе имаме $\binom{512}{1} = 512$, за 2 $\binom{512}{2} = 130816$, за 3 $\binom{512}{3} = 22238720$, а за 4 ќе имаме $\binom{512}{4} = 2829877120$ вредности за $u$, т.е. пресметувања на $a_u$. Освен тоа, секое пресметување на $a_u$ се состои од $2^k$ собироци, каде $k$ е саканиот степен.

[22] threshold value

4. Ако $D^2 > x_\alpha$ мораме да ја отфрлиме хипотезата $H_0^d$, т.е. хаш функцијата покажала статистички нестабилности[23].

Ќе вршиме два теста, $T_1^d$ и $T_2^d$.

За $T_1^d$, избираме $\nu = m - 1$ (бројот на излезни ANF). $n_i^d$ е очекуваната вредност од теоремата 2.6.1, која е иста за секое $i$. $\hat{n}_i^d$ е посматраниот број на мономи со степен $d$.

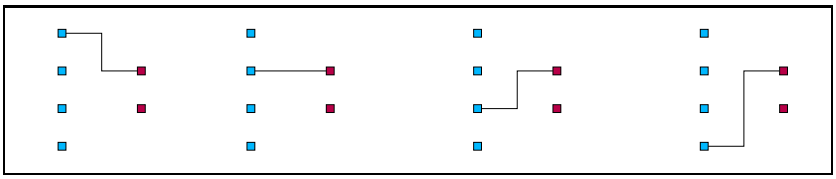
Во $T_2^d$, излезните ANF ги групираме во $2 \leq \nu \leq 9$ класи. Во оригиналниот текст не се дава никакво друго ограничување на критериумот за избор на класи, освен тоа дека математичкото очекување на бројот на елементи во една класа треба да биде поголемо од 5. Сепак, по детално разгледување на тестовите што самиот автор ги изведувал на хаш функции (коешто може да се најде на веб страните наведени во оригиналниот текст), ние го следиме неговиот пример, и за критериум на избор на класи земаме да има 3 класи такви што, на пример за хаш функција со 160-битен излез, се очекува во првата и третата класа да влезат по 30 ANF, а во втората класа да влезат 100 ANF. За повеќе информации, и поинакви конструкции на класите треба да се види придружниот Maple документ anf.mws.

Последниве математички очекувања, 3 на број, одговараат на $n_d^1, n_d^2, n_d^3$, а бројот на ANF кои навистина ќе влезат во секоја од класите ќе биде $\hat{n}_d^1, \hat{n}_d^2, \hat{n}_d^3$.

И во двата теста случајните променливи $n_d^i$ имаат нормална распределба, па според тоа случајната променлива $D^2$, која е нивна сума, има $\chi^2$-распределба со степени на слобода за еден помал од бројот на променливи во сумата; што го објаснува чекорот 4.

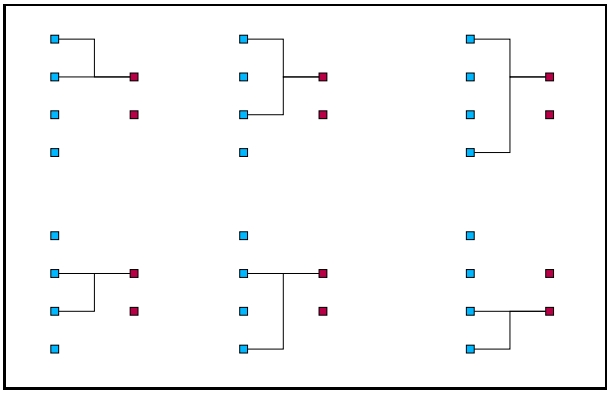
Намената на $T_1^d$ тестот е да даде (локална) слика за слабостите на поединечните ANF од излезот, додека намената на $T_2^d$ тестот е да даде глобална слика, со која што може да се види дали локалните слабости имаат глобално значење.

Зависноста меѓу влезните и излезните битови, т.е. Булови функции, т.е. нивните ANF, може сликовите да се прикаже со следниве едноставни примери, за хаш функција $h : B_2^4 \to B_2^2$:

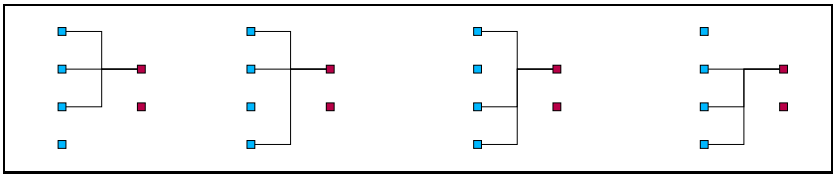


Сл. 2.1: зависности кај ANF со степен 1

[23] biases



Сл. 2.2: зависности кај ANF со степен 2



Сл. 2.3: зависности кај ANF со степен 3

# Глава 3

# Пакетот hashtest

## 3.1 Цели

Задача на дипломската работа беше да се организира достапното знаење во врска со тестирањето на хаш функции и да се изработи софтвер што ќе служи како алатка за тестирање и помагало при развој на истите. Дизајнот на софтверот, освен врз теориските основи дадени во претходниот дел, е заснован и врз следниве практични цели:

**Да биде проширлив.** Со минимален напор, т.е. со минимални познавања на програмирање во C и компајлери, да може да се проширува и да се користи за испитување нови (имплементации на) хаш функции. Резултат на тоа е архитектурата на plugin-и. Софтверот нема потреба (иако лесно може) да се прекомпајлира. Секоја хаш функција што се сака да се тестира се дефинира преку една динамичка деллива библиотека (DLL), на која ѝ е наметнато единственото ограничување да биде вклопена во потребниот интерфејс. Тест програмите потоа *при извршување (run-time)* ја вчитуваат библиотеката и ги вршат своите операции. На овој начин се остава простор за произволна флексибилност при имплементирањето на најкритичниот дел – самите хаш функции. Така, тие можат да се пишуваат и во друг програмски јазик, на пример, во ASM за да се постигне максимална ефикасност.

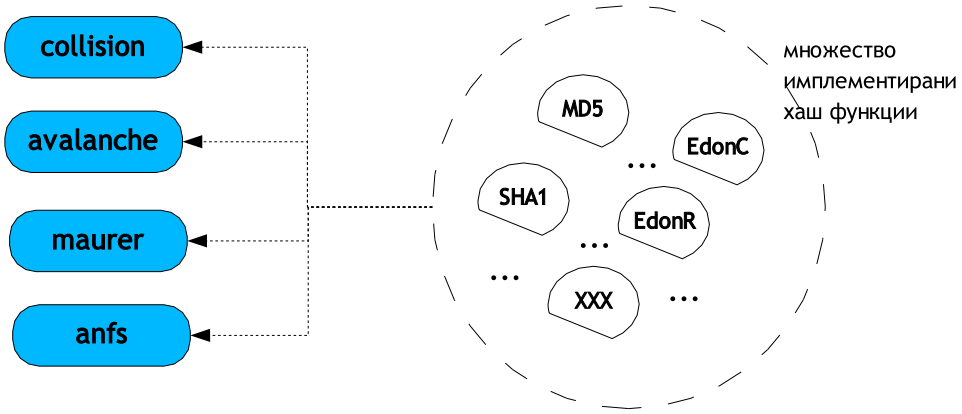
**Да дава релевантни резултати.** Важно е да се демонстрира дека некој систем навистина работи. Затоа се вклучени и имплементации на познати (MD5, SHA1) и помалку познати (EdonC, EdonR) хаш функции. Резултатите што ги даваат програмите за тестирање можат да се споредат со веќе познатите, објавени анализи. Едно испитување е дадено и во Додатокот A.

**Да биде ефикасен.** Користени се најдобрите можни алгоритми и процесор-меморија баланси. Пишуван е во стандарден C, а компајлиран со Pentium Pro оптимизации.

**Да биде преносliв.** До крајни граници е тестиран и за Windows и за GNU, така што е компајлиран и со GCC и со MSVC компајлерот. Сепак,

софтверот нема да се извршува на GNU, затоа што потсистемите за вчитување на динамички библиотеки помеѓу овие 2 типа оперативни системи се различни. Доколку се укаже потреба, со минимален труд, би можел да се пренесе и на GNU.

## 3.2 Архитектура




**buildall.bat** скрипта за компајлирање на сите тестови и вклучени хаш функции

**defs.h** глобални дефиниции

**test/** директориум што ги содржи имплементациите на тестовите

- **collision.c** тест за барање (делумни) колизии
- **avalanche.c** тест за својството на лавина
- **maurer.c** Мауреровиот статистички тест
- **anfs.c** Möbius-ова ANF анализа
- **bitflip.c** рутини за манипулација со битови што ги користат повеќето тестови
- **anfsdefs.c** константи потребни за anfs.c

**hash/** директориум што содржи имплементации на некои хаш функции

- **hash_md5.c** имплементација на MD5 од [11]
- **hash_sha1.c** имплементација на SHA1 од [11]
- **hash_edonc.c** имплементација на EdonC од [12]
- **hash_edonr.c** имплементација на EdonR од [12]
- **hash.c** plugin поврзувања на тестовите со хаш функциите
- **hash.h** заглавје за hash.c
- **hash_interface.h** интерфејс што треба да го имплементира секоја хаш функција која треба да се тестира
- **md.h** дефиниции специфични за MD5 и SHA1
- **edon.h** дефиниции специфични за Edon

## 3.3 Упатство за користење

```
avalanche [seq|rho] iteracii hash.dll

collision [full|<bits>] hash.dll

maurer [short|normal|long] hash.dll

anfs [1|2|3] hash.dll
```

### 3.3.1 avalanche

```
avalanche [seq|rho] iteracii hash.dll
```

Програмата `avalanche` се користи за мерење на својството на лавина.

Прошетката низ просторот на хаш функции може да се врши на 2 начина:

**seq** секвенцијално, кога следната вредност за оригинал е претходната зголемена за 1 како број; дава повеќе локална слика;

**rho** со $\rho$-методот, кога следната вредност за оригинал се формира од моменталната хаш вредност; дава слика што е порамномерно распределена низ просторот на оригинали.

Параметарот `iteracii` го определува обемот на статистичкиот примерок.

Пример за ова тестирање:

```
C:\hashtest>avalanche rho 2000 md5

2000-iterations Avalanche Property Test on:

The MD5 hash/compression function, as implemented in gnupg.
identifier = md5-compress
input bits = 512
output bits = 128

0%  10%  20%  30%  40%  50%  60%  70%  80%  90%  100%
..................................................

One bit change of the source resulted in
mean=63.994563 and variance=4127.309872
bit changes of the hash value.

The minimum number of bit changes was 36,
and the maximum was 96, out of 128 total.
```

### 3.3.2 collision

```
collision [full|<bits>] hash.dll
```

Програмата `collision` се користи за барање целосни (параметар `full`) или делумни (параметар `<bits>`) колизии. Делумна колизија се јавува кога две хаш вредности имаат Хамингово растојание поголемо од одреден број битови.

Претходно се пушта `avalanche` и се забележува кој е максималниот број на бит-промени. До таа вредност, колизии би требало многу брзо да се најдат. На пример, за SHA1 по 1000 rho-итерации максимумот е 107. Тоа го земаме предвид во примеров, за `collision` да заврши во разумно време.

```
C:\hashtest>collision 110 sha1

110-BIT Collision Search for:

The SHA1 hash/compression function, as implemented in gnupg.
identifier = sha1-compress
input bits = 512
output bits = 160

Search starting... (press CTRL+C to abort)...

Found a 110 bit collision after 383354 iterations!

  x1 :3BDAC05CE4FF37CA4DC2493A98494077717E5370000000000000000000000000000000000000000000000000000000000000
h(x1):CD64E38401E8BB9B21A619B56D7B6110463BC983

  x2 :54F04C13BF5743BE3C1DBF09D02A7CCF7D3C79FA0000000000000000000000000000000000000000000000000000000000000
h(x2):22971E2F5226D4A65FD9E34EBE54932733E50F45
```

Оригиналите и нивните (делумни) колизии се дадени во хексадецимална форма заради прегледност. Ако се развијат во бинарна форма, ќе биде можно лесно да се забележи кои битови се разликуваат.

### 3.3.3 maurer

```
maurer [short|normal|long] hash.dll
```

Програмата `maurer` изведува Маурерова универзална статистичка анализа на примерок добиен со обем одреден од параметарот. Освен иницијализирачките $Q = 655360$ итерации, во зависност од параметарот, се изведуваат:

**short** 6 553 600 итерации, што е под препорачаната должина; оваа должина служи само за прелиминарни тестирања што не треба да земаат многу време;

**normal** 65 536 000 итерации, што е препорачан минимум за должината на низата што се тестира;

**long** 655 360 000 итерации, што изведува солиден тест на случајност; прашање е дали е оправдано да се изведуваат (толку или) повеќе итерации, заради нумеричките грешки кои се натрупуваат во секој пресметувачки процес со ограничена прецизност.

```
C:\hashtest>maurer normal md5

NORMAL (66191360 iterations) Maurer Universal Statistical Test on:

The MD5 hash/compression function, as implemented in gnupg.
identifier = md5-compress
input bits = 512
output bits = 128

0%  10%  20%  30%  40%  50%  60%  70%  80%  90%  100%
..................................................

X_u statistics = 15.167458 (15.167379 = theoretical math. expect. for uniform)

PASSES for alpha = 0.100000
PASSES for alpha = 0.050000
PASSES for alpha = 0.025000
PASSES for alpha = 0.010000
PASSES for alpha = 0.005000
PASSES for alpha = 0.002500
PASSES for alpha = 0.001000
PASSES for alpha = 0.000500
```

Резултат е вредноста на $X_u$ статистиката. Во заградите е дадено математичкото очекување на рамномерно распределена низа. Резултатот за $X_u$ статистиката на крајот се користи за да се дадат резултатите од тестирањето на хипотезата дали низата е рамномерно распределена, за двостран тест, со неколку нивоа на значајност.

### 3.3.4 anfs

```
anfs [1|2|3] hash.dll
```

Откако со Мауреровиот тест ќе се утврди дали хаш функцијата генерира низа независни и рамномерно распределени случајни променливи, т.е. индивидуални излезни битови, Булови функции, може да се изврши Мобиусова статистичка анализа со помош на програмата `anfs`.

Параметар на тестот е степенот на ANF. Претходно се објаснети практичните причини заради кои степенот може да биде 1, 2 или 3 – со негово линеарно зголемување, временската комплексност на пресметувањето се зголемува експоненцијално.

```
C:\hashtest>anfs 2 sha1.dll

DEGREE=2 Mobius ANF Analysis on:

The SHA1 hash/compression function, as implemented in gnupg.
identifier = sha1-compress
input bits = 512
output bits = 160

0%  10%  20%  30%  40%  50%  60%  70%  80%  90%  100%
.................................................

130816 iterations; 523264 hash calculations

T_1^2 D^2==70.897581
    chi^2==189.424220l  =>  PASSED for significance alpha==0.05
    chi^2==203.399752l  =>  PASSED for significance alpha==0.01
    chi^2==219.846046l  =>  PASSED for significance alpha==0.001

T_2^2 D^2==0.426667
    chi^2==5.991465l  =>  PASSED for significance alpha==0.05
    chi^2==9.210340l  =>  PASSED for significance alpha==0.01
    chi^2==13.815511l  =>  PASSED for significance alpha==0.001
```

Резултат се двата теста, $T_1^d$ и $T_2^d$, интерпретирани за различни нивоа на значајност, за дадениот степен $d \in \{1, 2, 3\}$.

## 3.4 Упатство за проширување со нови хаш функции

**Кодирање.** Изворниот код за новата хаш функција (нека се вика `novhash.c`) се копира во директориумот `hash/`, и на почетокот на `novhash.c` се додаваат линиите:

```
#include "../defs.h"
#include "hash_interface.h"
```

На кодот не му е наложено никакво друго ограничување, освен што се бара да бидат имплементирани и следниве функции:

```
extern __declspec(dllexport) u16 hash_input_bit_length();
```

која го враќа бројот на влезни битови на хаш функцијата.

```
extern __declspec(dllexport) u16 hash_output_bit_length();
```

која го враќа бројот на излезни битови на хаш функцијата.

```
extern __declspec(dllexport) char *hash_name();
```

која враќа подолг текстуален стринг што ја идентификира хаш функцијата.

```
extern __declspec(dllexport) char *hash_description();
```

која враќа краток текстуален стринг што ја опишува хаш функцијата.

```
extern __declspec(dllexport) void hash_startup();
```

процедура што сите тест програми ја извршуваат еднаш, при иницијализирање на тестирањата. Ако нема потреба од такво нешто, тогаш таа треба да се имплементира како празна функција.

```
extern __declspec(dllexport) void hash_shutdown();
```

процедура што сите тест програми ја извршуваат еднаш, при завршување на тестирањата. Повторно, ако нема потреба од такво нешто, тогаш таа треба да се имплементира како празна функција.

```
extern __declspec(dllexport) int hash_transform(byte*,byte*);
```

која претставува едно пресметување на хаш вредност на првиот параметар. Хаш вредноста се сместува во вториот параметар. Функцијата враќа 1 ако сé било во ред, или 0 ако се случила некаква грешка.[1]

Во пакетот се вклучени четири имплементации на хаш функции, и тие можат да се искористат за програмирање по аналогија, зашто од нив може убаво да се види како овие функции се имплементираат.

**Компајлирање.** По кодирањето, од `novhash.c` треба да се добие DLL датотека. Тоа се прави со Microsoft Visual C++ компајлерот, со следнава наредба:

```
cl /G6 /Ox /LD hash\novhash.c /Fenovhash.dll
```

или ако не се сака Pentium Pro код со оптимизации, едноставно:

```
cl /LD hash\novhash.c /Fenovhash.dll
```

При сето ова мора да се внимава на мали и големи букви во параметрите, оти компајлерот е чувствителен на тоа.

Како што веќе објаснивме, се работи за плагин архитектура, и програмите за тестирање нема потреба да се прекомпајлираат. Едноставно, тестовите се извршуваат со параметар `novhash.dll`.

Ако сепак постои желба сите тестови и приложени хаш функции да се искомпајлираат од почеток, тоа може да се направи со (модификација на) скриптата `buildall.bat`.

[1] Во моменталната имплементација на програмите за тестирање овој резултат не се зема предвид, за да се запази едноставноста на кодот. Справувањето со грешки во C гарантирано го комплицира изворниот код, а од тоа во тестовите и нема потреба.

# Додаток А

# Резултати од некои тестирања

**Својство на лавина за 128-битни хаш функции по 5000 $\rho$-итерации**

| хаш функција | просек | дисперзија | min | max |
|---|---|---|---|---|
| MD5 | 63.999195 | 4127.919100 | 36 | 96 |
| EdonC 128-bit | 63.998888 | 4127.829649 | 36 | 92 |
| EdonR 128-bit | 63.994578 | 4127.342406 | 33 | 92 |

**Својство на лавина за 160-битни хаш функции по 5000 $\rho$-итерации**

| хаш функција | просек | дисперзија | min | max |
|---|---|---|---|---|
| SHA-1 | 80.000848 | 6440.133338 | 50 | 110 |
| EdonC 160-bit | 80.000309 | 6440.066016 | 49 | 110 |
| EdonR 160-bit | 79.992679 | 6438.845868 | 51 | 111 |

**Потребни итерации за да се дојде до делумни колизии, за 128-битни хаш функции**

| битови | 90 | 91 | 92 | 93 | 94 | 95 | 96 |
|---|---|---|---|---|---|---|---|
| MD5 | 181921 | | 274830 | 45 101 984 | 79 514 544 | | 81 343 278 |
| EdonC 128-bit | | 262813 | 1570617 | | | 4 883 355 | |
| EdonR 128-bit | 465883 | | 677348 | | | 7 515 481 | |

**Потребни итерации за да се дојде до делумни колизии, за 160-битни хаш функции**

| битови | 107 | 109 | 110 | 111 | 112 | 113 | 118 |
|---|---|---|---|---|---|---|---|
| SHA-1 | | 82432 | 383354 | 1 449 009 | 2 259 464 | 13 135 456 | 14 044 770 |
| EdonC 160-bit | | 105209 | 204901 | 1 391 314 | | 2 143 924 | |
| EdonR 160-bit | 62171 | | | 372542 | | 3 432 190 | |

**Maurer-ов тест за униформност**

| | |
|---|---|
| MD5 | 15.167458 |
| EdonC 128-bit | 15.167506 |
| EdonR 128-bit | 15.167343 |
| SHA-1 | 15.167256 |
| EdonC 160-bit | 15.167243 |
| EdonR 160-bit | 15.167153 |
| очекување за рамномерна распределба | 15.167379 |

**Möbius-ова статистичка анализа на 128-битни хаш функции**

| хаш функција | $T_1^1D^2$ | $T_1^2D^2$ | $T_1^3D^2$ | $T_2^1D^2$ | $T_2^2D^2$ | $T_2^3D^2$ |
|---|---|---|---|---|---|---|
| MD5 | 62.601563 | 68.744053 | 67.498870 | 1.654167 | 2.466667 | 0.554167 |
| EdonC 128-bit | 60.093750 | 53.710632 | 73.149739 | 3.000000 | 2.154167 | 4.554167 |
| EdonR 128-bit | 55.128906 | 67.455051 | 55.935845 | 2.616667 | 2.887500 | 3.320833 |
| 0.05 | 154.301516 | | | 5.991465 | | |
| 0.01 | 166.987390 | | | 9.210340 | | |
| 0.001 | 181.993045 | | | 13.815511 | | |
| значајност $\alpha$ | $\chi^2_{127}$ распределба | | | $\chi^2_2$ распределба | | |

**Möbius-ова статистичка анализа на 160-битни хаш функции**

| хаш функција | $T_1^1D^2$ | $T_1^2D^2$ | $T_1^3D^2$ | $T_2^1D^2$ | $T_2^2D^2$ | $T_2^3D^2$ |
|---|---|---|---|---|---|---|
| SHA1 | 76.878906 | 70.897581 | 89.457610 | 0.256667 | 0.426667 | 2.016667 |
| EdonC 160-bit | 85.480469 | 75.221685 | 81.467442 | 0.443333 | 1.560000 | 3.693333 |
| EdonR 160-bit | 102.281250 | 77.200511 | 90.228846 | 2.016667 | 3.266667 | 4.843333 |
| 0.05 | 189.424220 | | | 5.991465 | | |
| 0.01 | 203.399752 | | | 9.210340 | | |
| 0.001 | 219.846046 | | | 13.815511 | | |
| значајност $\alpha$ | $\chi^2_{159}$ распределба | | | $\chi^2_2$ распределба | | |

# Литература


[1] Eric Filiol: *A New Statistical Testing for Symmetric Ciphers and Hash Functions*, ESAT - Virology and Cryptology Lab, 2002
`http://www-rocq.inria.fr/codes/Eric.Filiol/Moebius/index.html`

[2] Ueli M. Maurer: *A Universal Statistical Test for Random Bit Generators*, Department of Computer Science, Princeton University

[3] Paul C. van Oorschot & Michael J. Wiener: *Parallel Collision Search with Cryptanalytic Applications*, Nortel, Canada; 1996

[4] Alfred J. Menezes, Paul C. van Ooorschot & Scott A. Vanstone: *Handbook of Applied Cryptography*, CRC Press, 5th printing, 2001;
`http://www.cacr.math.uwaterloo.ca/hac/`

[5] Douglas R. Stinson: *Cryptography: Theory and Practice*, CRC Press, 1995

[6] Michael J. Wiener: *The Full Cost of Cryptanalitic Attacks*

[7] Donald E. Knuth: *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*, Addison-Wesley, 3rd edition, 1997

[8] Donald E. Knuth: *The Art of Computer Programming, Volume 3 / Sorting ans Searching*, Addison-Wesley, 2nd edition, 1998

[9] D. Gligoroski, S. Markovski and V. Bakeva: *"Edon" – C and R, Two infinite classes of strongly collision resistant hash functions with variable length of output*, Институт за информатика, draft version 0.992, January 2003

[10] Stephen Cook: *The P versus NP Problem*, University of Toronto

[11] Имплементацијата на MD5 и SHA1 е земена од пакетот GPG (The GNU Privacy Guard) кој се дистрибуира како слободен софтвер;
`http://www.gnupg.org/`

[12] Имплементацијата на Edon хаш функциите е земена од референтната имплементација достапна на
`http://www.pmf.ukim.edu.mk/~danilo/`