MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Automatické hľadanie závislostí v kandidátnych hašovacích funkciách SHA-3

BAKALÁRSKA PRÁCA

Ondrej Dubovec

Brno, jaro 2012

Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Vedúci práce: RNDr. Petr Švenda, Ph.D.

Poďakovanie

Ďakujem vedúcemu práce RNDr. Petrovi Švendovi, Ph.D. za odborné vedenie práce, cenné rady a čas ktorý mi venoval. Poďakovanie patrí aj kolegovi Bc. Matejovi Prišťákovi za spoluprácu pri úpravách použitej aplikácie, ako aj celej mojej rodine za podporu.

Zhrnutie

Práca je zameraná na hľadanie závislostí v kandidátnych hashovacích funkciách SHA-3 pomocou nástroja využívajúceho technológiu evolučného obvodu. Vybrané funkcie boli po oslabení bezpečnostných parametrov podrobené testom na vyhľadávanie nežiadúcich závislostí a to viacerými spôsobmi.

Kľúčové slová

hashovacie funkcie, kolízny útok, SHA-3 kandidáti, evolučný algoritmus, evolučný obvod, Sensor Security Simulator

Obsah

1 **Úvod** . . . 1
2 **Hashovacie funkcie** . . . 3
2.1 *Základné vlastnosti a definície* . . . 3
2.1.1 Jednosmerná hashovacia funkcia (OWHF) . . . 5
2.1.2 Bezkolízna hashovacia funkcia (CRHF) . . . 5
2.1.3 Univerzálna jednosmerná hashovacia funkcia (UOWHF) . . . 5
2.1.4 Autentizačný kód správy (MAC) . . . 6
3 **Útoky na hashovacie funkcie** . . . 7
3.1 *Základné rozdelenie útokov* . . . 7
3.1.1 Útoky na MDC . . . 7
3.1.2 Útoky na MAC . . . 8
3.2 *Narodeninový útok* . . . 8
3.3 *Rainbow tables* . . . 9
3.4 *Praktický útok na MD5* . . . 10
4 **Použité nástroje a evolúcia** . . . 12
4.1 *Evolučné a genetické algoritmy* . . . 12
4.2 *Evolučné obvody, Sensor Security Simulator* . . . 13
4.3 *Náhodné dáta* . . . 15
4.4 *Implementácia kandidátnych funkcií* . . . 16
4.5 *Priebeh testov* . . . 16
5 **Spôsoby testovania a výsledky** . . . 18
5.1 *Rozlišovanie pri obmedzenom vstupe funkcie* . . . 18
5.1.1 Testovacie vektory . . . 18
5.1.2 Vstupy obvodu . . . 19
5.1.3 Výstupy obvodu . . . 19
5.1.4 Spôsoby predikcie . . . 19
5.1.5 Rozšírenie vstupov obvodu . . . 19
5.1.6 Nastavenie evolúcie a obvodu . . . 20
5.1.7 Testy na funkcii MD5 . . . 20
5.1.8 Výsledky pre SHA-3 kandidátov . . . 21
5.2 *Odhad obmedzenej časti vstupných dát* . . . 25
5.2.1 Testovacie vektory . . . 25
5.2.2 Vstupy obvodu . . . 25
5.2.3 Výstupy obvodu . . . 25
5.2.4 Spôsob predikcie . . . 26
5.2.5 Nastavenie evolúcie a obvodu . . . 26
5.2.6 Výsledky pre SHA-3 kandidátov . . . 26
5.3 *Avalanche efekt* . . . 27
5.3.1 Testovacie vektory . . . 27
5.3.2 Vstupy obvodu . . . 27

5.3.3 Výstupy obvodu 27
5.3.4 Spôsob predikcie 28
5.3.5 Nastavenie evolúcie a obvodu 28
5.3.6 Výsledky pre SHA-3 kandidátov 28
6 **Záver** 30
A **Grafy** 31
Bibliografia 39

Kapitola 1

Úvod

Bezpečnosť bola jedným z najdôležitejších aspektov pri komunikácii už v dávnej minulosti. Vždy bolo potrebné dodanie nepoškodenej správy, poprípade zabezpečiť, aby správu dokázal prečítať len pravý príjemca. Takto vznikali rozličné techniky, ako napríklad napísanie správy na oholenú hlavu otroka, ktorý ju doručil až po opätovnom narastení vlasov, kde bolo cieľom zabrániť náhodnému človeku získať prepravovanú správu. Správa mohla byť počas prepravy úmyselne, či neúmyselne zmenená. Vyriešiť to mohli napríklad používané pečate, ktoré zabezpečovali „podpis“ a teda overenie pôvodu správy. Spomínané techniky však nepredpokladali znalosť týchto postupov u možných útočníkov.

Techniky sa neustále zlepšovali, no v posledných desaťročiach problém nabral nové rozmery. Rapídny rozvoj informačných technológií, verejne prístupný Internet a mnohé ďalšie aspekty zvýšili potrebu bezpečnosti a zabezpečenia dát. Je nutné poskytnúť bezpečnú autentizáciu pri prihlasovaní na akýkoľvek server. Zabezpečenie internetového bankovníctva, ktoré zaručuje, že prevod vykonávame skutočne na účet, ktorý sme zadali. Poskytovanie istoty, že dáta sú nám prezentované v nezmenenej podobe od autora, ktorému dôverujeme. Preto je nutné zaistiť autenticitu, integritu, nepopierateľnosť dát, kde využívame techniky digitálnych podpisov, hashovania a šifrovania.

V súčastnosti medzi používané kryptografické hashovacie funkcie (ďalej označované ako hashovacie funkcie, pokiaľ nieje explicitne uvedené inak) patria funkcie MD5, SHA-1 a SHA-2. MD5 je stále rozšírená v mnohých systémoch a to aj napriek odporúčaniu funkciu nepoužívať z dôvodu jej prelomenia v roku 2004. Funkcia SHA-1, ktorá vychádza z pôvodnej SHA-0 bola taktiež prelomená v roku 2005 publikovaným útokom, ktorý nachádzal kolízie v čase lepšom ako v prípade útoku hrubou silou [4]. Očakávanie prelomenia funkcie SHA-2 sú pravdepodobne jedným z dôvodov vypísania súťaže o novú hashovaciu funkciu, ktorá ponesie názov SHA-3.

Prvá časť bakalárskej práce sa zaoberá všeobecným pohľadom na hashovacie funkcie, popisuje ich hlavné rozdelenie, v krátkosti približuje aj funkcie, ktoré nie sú predmetom skúmania tejto práce a objasňuje ich využitie v praxi. V podkapitolách je podrobne rozobraté rozdelenie kryptografických hashovacích funkcií, ako aj bezpečnostné požiadavky kladené na každý typ funkcie. Práca pokračuje popisom všeobecných aj viac konkretizovaných útokov na hashovacie funkcie. Taktiež je v krátkosti popísaná súťaž o SHA-3 algoritmus, vymenované funkcie, ktoré sa do súťaže zapojili a ktoré z nich sú hlavným cieľom nášho testovania.

Nasledujúca kapitola spolu s podkapitolami popisuje technológiu evolučných obvodov,

ich funkcie a tiež ich implementáciu v aplikácii Sensor Security Simulator. Je tu predstavené použitie obvodu na minulé výpočty, teda napr. pokusy o hľadanie inverzie u funkcií MD5 a SHA-1. Kapitola neskôr popisuje nami navrhnuté postupy testovania kandidátnych funkcií, ich konkrétne riešenia v evolučnom obvode a implementáciu v pôvodnej aplikácii. Tiež popisujeme spôsob integrácie SHA-3 kandidátov do projektu podľa predpísaného NIST rozhrania, ktoré museli kandidátne funkcie spĺňať.

Posledná kapitola rozoberá jednotlivé testované algoritmy a prezentuje získané výsledky. Uvádzame nastavenia obvodu počas testov, výsledné hodnoty jednotlivých testov, porovnávame grafy behov a obvody, ktoré boli schopné sa na dátach učiť.

Text bakalárskej práce bol vytvorený pomocou značkovacieho jazyku DocBook vo formáte XML. Následne boli na prevod použité XSL transformácie od Jana Pavloviča [2]. Na generovanie grafov evolučných obvodov bol použitý software Graphviz [7].

Kapitola 2

Hashovacie funkcie

Kryptografické hashovacie funkcie tvoria nenahraditeľnú časť súčasnej kryptografie. Úlohou všetkých hashovacích funkcií je prevod vstupných dát rôzneho formátu a ľubovoľnej, ale konečnej dĺžky (správy, message) na výstup fixnej dĺžky, označovaný ako hash (hash value, message digest, digest), ktorý nám slúži ako kompaktná reprezentácia vstupných dát. Aby však funkcie využívané v oblasti bezpečnosti boli schopné plniť svoju úlohu, kladieme dodatočné požiadavky na vlastnosti ich výstupu, čím nám vznikajú nástroje vhodné pri aplikáciách požadujúcich zvýšenú úroveň bezpečnosti.

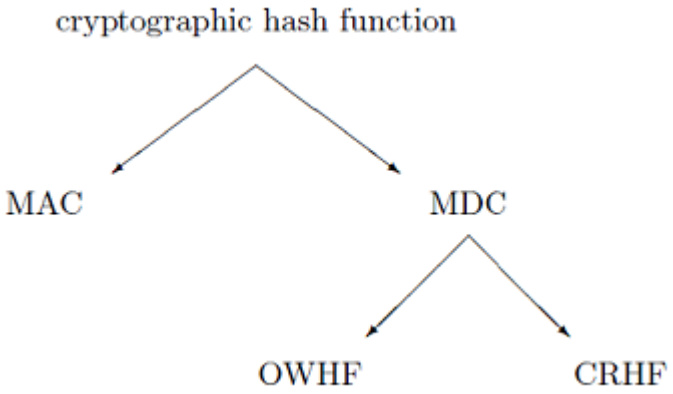
Z dôvodu zabezpečenia integrity dát využívame hashovacie funkcie pri technikách digitálneho podpisu (digital signature). V tomto prípade je dokument ľubovoľnej dĺžky najprv hashovaný pomocou známej hashovacej funkcie a až výsledný hash následne podpísaný (zašifrovaný) privátnym kľúčom odosielateľa. Možnosť podpisu krátkeho hashu poskytuje výhodu oproti nutnosti podpisovať pôvodný dokument, ktorého veľkosť sa može pohybovať v MB i GB. Nieje totiž bezpečné dokument rozdeliť na menšie časti a tie následne podpisovať. Takto podpísanú správu je následne možné zo strany príjemcu overiť, a to na základe znalosti použitej hashovacej funkcie a znalosti verejneho kľúča. Digitálny podpis okrem integrity zabezpečuje aj autenticitu a nepopierateľnosť pôvodu dokumentu.

Osobitnou triedou hashovacích funkcií sú Message Authentication Codes (MACs). Môžeme ich označiť ako kľúčované hashovacie funkcie (keyed hash functions). Ako vstup príjmajú dáta konečnej dĺžky a tajnú informáciu, ktorou je kľúč. Spomínaný kľúč musí byť vopred dohodnutým zdieľaným tajomstvom medzi odosielateľom a príjemcom, resp. medzi všetkými účastníkmi komunikácie, ako je to v prípade symetrickej kryptografie. Výstupom je reťazec fixnej dĺžky, pričom bez znalosti tajného kľúča je „obtiažne“ [1] nájsť rovnaký hash bez znalosti daného kľúča. MACs nám zabezpečujú integritu dát, ako aj overenie autenticity dát medzi vlastníkmi tajného kľúča. Funkcie typu MAC je možné skonštruovať z hashovacej funkcie (napríklad tajná časť hashu), alebo ľubovoľnej symetrickej šifry za predpokladu použitia režimu cipher block chaining [5].

2.1 Základné vlastnosti a definície

Základné rozdelenie hashovacích funkcií vyjadruje nasledujúci obrázok [3]

1. v angličtine označované slovným spojením „computationally infeasible“ - znamená možnosť riešenia problému len na teoretickej úrovni. Pri súčasnej dostupnej výpočetnej technike by praktické riešenie nebolo možné



Obrázok 2.1: Rozdelenie kryptografických hashovacích funkcií

Hashovacie funkcie delíme podľa obrázku (Obrázok 2.1) na Autentizačné kódy správ (Message Authentication Codes, MACs) a Manipulačné detekčné kódy (Manipulation Detection Codes, MDCs). Druhá kategória funkcií sa ďalej rozdeľuje podľa bezpečnostných požiadavkov a to nasledovným spôsobom:

- Jednosmerná hashovacia funkcia (One-way hash function, OWHF)
- Hashovacia funkcia odolná voči kolízii (Collision resistant hash function, CRHF)
- Univerzálna jednosmerná hashovacia funkcia (Universal one-way hash function, UOWHF)

Obecná hashovacia funkcia je funkcia h so vstupom X, ktorá spĺňa nasledovné podmienky:

- Jednoduchosť výpočtu - pomocou funkcie h je jednoduché vypočítať hodnotu h(X).
- Kompresia - vstup konečnej dĺžky X je pomocou funkcie h mapovaný na výstup h(X) o fixnej dĺžke n bitov.

Podľa typu funkcie požadujeme dodatočné bezpečnostné vlastnosti. Tieto vlastnosti sú:

- Odolnosť voči nájdeniu vzoru (preimage resistance).
- Odolnosť voči nájdeniu druhého vzoru (second-preimage resistance).
- Odolnosť voči kolíziám (collision resistance).

---

dosiahnuť v konečnom čase.

2.1.1 Jednosmerná hashovacia funkcia (OWHF)

Jednosmerná hashovacia funkcia h (one-way, hash function, weak one-way hash function) je funkcia (podľa definície z [3]) spĺňajúca nasledujúce kritériá:

- Popis funkcie h musí byť prístupný verejnosti a nesmie obsahovať žiadnu formu utajenej informácie potrebnú pre činnosť. Táto vlastnosť vychádza z Kerckhoffovho princípu [1], ktorý hovorí, že bezpečnosť systému zostáva zachovaná aj v prípade, že algoritmus je známy širokej verejnosti. Jedinou utajovanou informáciou je teda kľúč.
- Funkcia spĺňa vlastnosti obecnej hashovacej funkcie, ktorými sú kompresia a jednoduchosť výpočtu [2.1]. Dĺžka výstupu n navyše spĺňa podmienku ($n \geq 64$).
- Pre pevne daný hash Y je „obtiažne“ nájsť správu X, pre ktorú platí Y = h(X).
- Ku zadanej správe X je „obtiažne“ nájsť správu X′ takú, že $X \neq X'$ a h(X) = h(X′).

Tretia vlastnosť OWHF, taktiež označovaná ako jednosmernosť (one-wayness, preimage resistance) zabezpečuje funkcii odolnosť voči útokom typu preimage. Posledná vlastnosť sa tiež nazýva slabá bezkolíznosť (weak collision resistance, second-preimage resistance), je striktnejšia a relevantná pre využitie v reálnych aplikáciách.

Jednoduchým príkladom funkcie, ktorá nespĺňa jednosmernosť môže byť násobenie. Ľahko totiž vieme k funkcii nájsť inverziu. Funkciou, ktorá jednosmerná je, môže byť napríklad faktoriál čísla.

2.1.2 Bezkolízna hashovacia funkcia (CRHF)

Funkciu h označujeme ako bezkolíznu (collision resistant, niekedy aj strong one-way hash function) pokiaľ má nasledujúce vlastnosti [3]:

- Funkcia spĺňa všetky požiadavky kladené na jednosmernú hashovaciu funkciu podľa [2.1.1]. Dĺžka výstupu n navyše spĺňa podmienku ($n \geq 128$).
- Je „obtiažne“ nájsť 2 správy X, Y, pre ktoré platí $X \neq Y$ a h(X) = h(Y).

Pridaná vlastnosť funkcie značí bezkolíznosť (odolnosť voči kolíznym útokom, collision resistance). Ako však uvádza definícia, vlastnosť zaručuje obtiažny nález kolízie, ale možnosť existencie kolízie nijak nevylučuje. Uvážme hashovaciu funkciu podľa definície [2.1.2] s výstupom dĺžky 128 bitov, ktorá má teda k dispozícii 2^{128} možných hodnôt hashov. V prípade, keď uvážime vstupy ľubovoľnej dĺžky, je zrejmé, že funkcia nedokáže vyprodukovať unikátny hash pre každý z týchto vstupov.

2.1.3 Univerzálna jednosmerná hashovacia funkcia (UOWHF)

Koncept UOWHF bol prvý raz prezentovaný v publikácii M. Naora a M. Yunga [14]. Podstatou je používanie veľkej sady hashovacích funkcií namiesto jedinej. Funkcia je vybratá náhodne a nezávisle na type hashovaných dát, z čoho vyplýva, že hľadanie kolízie pre jedinú z nich je zbytočné.

2.1.4 Autentizačný kód správy (MAC)

Funkcia h je typu MAC, pokiaľ spĺňa nasledujúce vlastnosti [3]:

- Popis funkcie h musí byť prístupný verejnosti, kľúč je ale nutné udržať v tajnosti.
- Výsledok funkcie h(K,X) je fixnej dĺžky n (n $\geq$ 32...64), kde X je správa a K tajný kľúč.
- Pri zadanom h, X a K je výpočet h(K,X) jednoduchý.
- So znalosťou h a X je „obtiažne“ získať hodnotu h(K,X) s pravdepodobnosťou úspechu vyššou ako $1/2^n$.
- Aj v prípade, keď je známa veľká sada dvojíc {X_i, h(K,X_i)}, kde X_i je zvolené útočníkom, je „obtiažne“ získať kľúč K, alebo vypočítať h(K,X') pre každé $X' \neq X_i$.

Princíp funkcií typu MAC je podobný tým typu MDC, pribúda nám však navyše réžia správy a distribúcie tajného kľúča. Funkcia tiež spĺňa podmienky kladené na obecnú hashovaciu funkciu (jednoduchosť výpočtu a kompresia).

Kapitola 3

Útoky na hashovacie funkcie

3.1 Základné rozdelenie útokov

Cieľom útokov na hashovacie funkcie je využitie slabiny, ktorú by funkcia teoreticky obsahovať nemala. Preto sú útoky stavané hlavne na predpoklade, že funkcia nebude spĺňať niektorú z uvedených bezpečnostných požiadaviek a útočník sa ju pokúša odhaliť. Základné rozdelenie môže byť nasledovné:

- Útoky na MDC
- Útoky na MAC

3.1.1 Útoky na MDC

- (First-)preimage attack (vzorový útok) - podstata útoku je ku známemu hashu Y nájsť taký vzor (správu) X, pre ktorý bude platiť Y = h(X).
- Second-preimage attack (druhý vzorový útok) - ku známej správe X a hodnote hashu h(X) hľadáme správu X', pre ktorú platí $X \neq X'$, h(X) = h(X').

Úspech útoku je podmienený dĺžkou n výstupného hashu, tzn. komplexnosť je stanovená ako 2^n. To je tiež dôvodom obmedzenia, ktoré určuje minimálnu dĺžku hashu ako 80 bitov a ďalšie zvýšenie poskytuje vyššiu odolnosť. Pokial útok typu first-, či second-preimage nedokážeme vykonať za menej, než spomínaných 2^n operácií, je funkcia považovaná za odolnú voči preimage útokom. [2.1.1].

- Collision attack (kolízny útok) - snahou útočníka je nájsť 2 správy X a Y také, že $X \neq Y$ a h(X) = h(Y).
- Prefix collision attack (kolízny útok so zvoleným prefixom) - útočník vyberá 2 prefixy p1 a p2 také, že platí $p1 \neq p2$. a k týmto prefixom hľadá sufixy s1, s2, pre ktoré bude platiť h(p1.s1) = h(p2.s2), kde „.“ je operátor zreťazenia.

Úspešnosť kolíznych útokov je zo všeobecného hľadiska vyššia ako pri preimage útokoch, a to z dôvodu platnosti narodeninového paradoxu, ktorý vysvetľujeme ďalej v kapitole. Môžeme ich však aplikovať len na funkcie typu CRHF, keďže OWHF odolnosť voči kolíziám nemajú. Vyššia náchylnosť funkcií vyplýva z faktu, že na prelomenie funkcie stačí 2

$^{n/2}$ vykonaných pokusov, preto je minimálna požadovaná dĺžka hashu n = 160 bitov. Všetky vyššie uvedené typy útokov sa tiež nazývaju generické, pretože ich úspech závisí len na dĺžke hashu, nie však na d'alšej znalosti algoritmu.

3.1.2 Útoky na MAC

Generický útok na MAC bude vyzerať nesledovne:

- Za predpokladu znalosti X a hodnoty h(K,X) je útočník bez znalosti kľúča K schopný nájsť správu X' a hash h(K,X'), pre ktorú platí $X \neq X'$.

Negenerické útoky môžeme podľa dodatočných znalostí útočníka rozlišovať nasledovne [3]:

- Known plaintext attack - útočník pozná niekoľko správ (plaintext) a k nim odpovedajúce MACy.
- Chosen plaintext attack - útočník si sám zvolí sadu plaintext správ a následne tiež všetky MACy k týmto správam.
- Adaptive chosen plaintext attack - najobecnejší útok, pri ktorom si útočník vyberá správy a k nim získava odpovedajúce MACy. Každý nasledujúci výber je ovplyvnený výsledkom toho prechádzajúceho.

3.2 Narodeninový útok

Narodeninový útok (Birthday attack) ako najznámejší útok so zameraním na hľadanie kolízií využíva nasledujúce znalosti. V skupine 23 ľudí majú aspoň 2 vybraní ľudia spoločný dátum narodenia s pravdepodobnosťou vyššou ako 50%. Pokiaľ uvážime skupinu o počte 30 ľudí, pravdepodobnosť rovnakého javu prekročí dokonca 70%. Výsledné pravdepodobnosti sú omnoho vyššie, než by sa očakávalo, preto tento jav nazývame „narodeninový paradox“. Všeobecný narodeninový útok môže vyzerať nasledovne:

- Zvolíme si náhodne správy $X_1 ... X_n$.
- Pre zvolené správy vypočítame $h(X_1) ... h(X_n)$.
- Výsledné hashe porovnávame, ak nájdeme dvojicu zhodných, kolízia bola nájdená.

Jedným z prvých aplikovaných útokov bol Yuvalov narodeninový útok, ktorý je možné využiť na každú bezkľúčovú hashovaciu funkciu a jeho zložitosť je $O(2^{m/2})$ v závislosti na dĺžke výstupu n.

- Na začiatku zvolíme správu X_1 a podvodnú správu X_2
- Generujeme $t = 2^{n/2}$ správ X'_1, ktoré sú drobnými modifikáciami správy X_1.

- Každú z upravených správ hashujeme zvolenou funkciou h a získané hodnoty ukladáme, aby bolo neskôr možné medzi nimi rýchlo vyhľadávať (v čase O(t)).
- Pre správu X_2 generujeme jej drobné modifikácie X'_2, ich hashe $h(X'_2)$ a postupne porovnávame s hodnotou X'_1. Postup opakujeme, pokiaľ nieje nájdená zhoda, ktorú očakávame po porovnaní t hodnôt X'_2 (porovnávanie s hodnotami uloženými v tabuľke vyžaduje konštantný čas).
- Výsledkom útoku sú hodnoty X'_1 a X'_2 získané drobnými úpravami X_1 a X_2, pre ktoré platí $h(X'_1) = h(X'_2)$

3.3 Rainbow tables

Útok využíva metódu time-memory trade-off, prvý raz publikovanú Martinom Hellmanom [8], ktorý popísal jej aplikáciu v oblasti kryptografie. Metóda nám poskytuje možnosť urýchlenia kryptoanalýzy vďaka vopred predpočítaným hodnotám uloženým v pamäti.

Rainbow table (v podkapitole označená ako tabuľka) je tabuľka predpočítaných hodnôt, určená na získanie inverzie k hashu (získanie plaintextu z hodnoty hashu), v praxi zvyčajne využívaná ako rozšírenie slovníkového útoku pri lámaní hesiel, za prekpokladu obmedzenej dĺžky a využitia znakov v hesle. Útočník si vytvára tabuľku, kam ukladá vypočítané hashe pre bežne používané, alebo krátke heslá, plaintext k týmto heslám s usporiadaním podľa hashov. Následne po jeho získaní sme schopní tento hash v krátkom čase v tabuľke vyhľadať a tak k nemu získať zodpovedajúce heslo. Problém nastáva v prípade útoku na dlhšie heslá, čo spôsobuje nárast veľkosti tabuľky a zároveň nárast potrebnej pamäti. Hellman taktiež zaviedol používanie redukčnej funkcie r, ktorá v našom prípade mapuje hashe na plaintext hodnoty. Spôsob mapovania, a teda spôsob tvorby hesla z hashu je závislý na tom, pre aké heslá má útočník snahu tabuľku skonštruovať (výskyt a počet písmen, číslic, dĺžka hesla, atď.). Redukčná funkcia nám tým zaručuje výskyt hesiel s určenou charakteristikou. S pomocou tejto funkcie sa následne do tabuľky ukladá plaintext, na ktorý opakovane aplikujeme hashovaciu funkciu h a funkciu r.

Ako sme uviedli, rainbow table obsahuje plaintext heslá a ich hashe. Tie sú usporiadané vo formáte tzv. Rainbow chains (reťazcov), pričom každý z reťazcov začína heslom a pokračuje hashom získaným aplikáciou funkcie h na túto hodnotu. Nasleduje znovu heslo, získané aplikáciou funkcie r na predchádzajúci hash. Takto vytvorený reťazec je zakončený hashom, pričom v tabuľke je v skutočnosti uvedené len prvé heslo a posledný získaný hash. V prípade, že sa rozhodneme v každej iterácii redukcie použiť inú r funkciu, zabránime vzniku kolízií, ktoré spôsobujú zmenšenie spektra pokrytých hesiel a môžu znehodnotiť celé reťazce v tabuľke. Je možné tiež využiť niekoľko tabuliek namiesto jedinej, s tým, že r funkcie sa budú líšiť medzi jednotlivými tabuľkami.

Medzi ďalšie vylepšenia algoritmu patria napríklad tzv. rozlišovacie body (distinguished points). Spôsob, ktorý publikoval Rivest v praxi výrazne znižuje počet potrebných prístupov do pamäti počas kryptoanalýzy. Tieto „body“ reprezentujú kľúče, ktoré udávajú is-

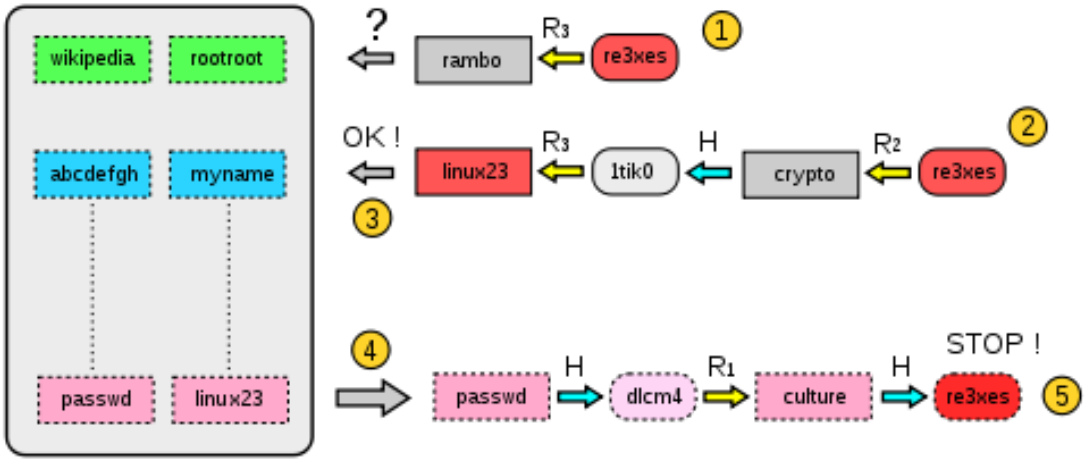
tú spoločnú vlastnosť (napríklad prvých 10 bitov je 0). Následne počas prístupu do pamäti kľúč vyhľadávame len v prípade, že danú vlastnosť splňuje. Vyhľadávanie v tabuľke môže vyzerať nasledovne:

1. Je zadaný hash h(x), pre ktorý hľadáme inverziu. Začíname prehľadávaním posledného stĺpca tabuľky, ktorý obsahuje finálne hodnoty reťazcov (hashe) a hľadáme zhodu.

2. V prípade, že zhodu hashov neobjavíme, používame redukčnú funkciu a jej výsledok hashujeme (ak sme pri konštrukcii tabuľky používali viac r funkcií, začíname tou poslednou). Znovu porovnávame hodnotu s posledným stĺpcom tabuľky.

3. V prípade objavenia zhody vieme, že nami hľadané heslo je súčasťou reťazcu v ktorom sme na zhodu narazili. Vezmeme prvú hodnotu reťazcu a opakovane aplikujeme hashovaciu a redukčnú funkciu. Akonáhle je hash lokalizovaný, hľadaným heslom je predchádzajúca hodnota v reťazci.

Nasledujúci obrázok prebraný z [10] ukazuje úspešné nájdenie hesla k zadanému hashu:



Obrázok 3.1: Vyhľadávanie hesla v rainbow table s konečným úspechom

Účinnou obranou proti útokom pomocou Rainbow tables je solenie hesiel. Soľ je dodatočná informácia, náhodne generovaná a pridávaná ku heslu pred tým, než je hashované. Dostatočne veľká a unikátna hodnota soli pre každého užívateľa nám zabezpečuje, že i rovnaké heslá budú mať rozdielne hodnoty hashov.

3.4 Praktický útok na MD5

Prvý z praktických útokov na funkciu MD5 bol publikovaný tímom X. Wangovej [15]. Ide o kolízny útok schopný rádovo v čase jednej hodiny nájsť 2 správy s dĺžkou 1024 bitov,

produkujúce totožný hash. Prvým krokom, ktorý zaberá väčšinu času útoku je nájdenie dvoch polspráv o dĺžke 512 bitov M_1 a M_2, medzi ktorými je nasledujúci vzťah:

- $M_2 = M_1 + C_1$, kde C_1 je 512 bitová konštanta, ktorá obsahuje tri zo šestnástich 32-bitových podblokov s nenulovou hodnotou.

Následne hľadáme polsprávy N_1 a N_2, čo sme schopní dosiahnuť v čase od 15 sekúnd do 5 minút:

- $N_2 = N_1 + C_2$, kde C_2 je 512 bitová konštanta, ktorá obsahuje tri zo šestnástich 32-bitových podblokov s nenulovou hodnotou a nie je zhodná s N_1.

Vo výsledku tak dostávame správy (M_1,N_1) a (M_2,N_2), pre ktoré platí h(M_1,N_1) = h(M_2,N_2). Útok bol publikovaný s použitím pôvodných iniciálnych hodnôt hashovacej funkcie. Autori však uvádzajú, že funguje nezávisle na ich hodnotách. Princíp generovania správ M a N však zverejnený nebol.

O rok neskôr bol publikovaný útok V. Klímy, ktorý vychádzal z predchádzajúceho útoku a tiež z analýz, ktoré sa pokúšali odhaliť algoritmus generovania správ M a N. Útočník postupoval podobným spôsobom hľadania polspráv $M_{1,2}$ a $N_{1,2}$, pričom prvá fáza prebiehala 1000-2000 krát rýchlejšie, druhá ale 2-80 krát pomalšie, autor tak uvádza 3-6 krát vyššiu rýchlosť vyhľadania kolízie ako u prechádzajúcej metódy [9].

Kapitola 4

Použité nástroje a evolúcia

V nasledujúcej časti práce popisujeme spôsob testovania kandidátnych funkcií, v krátkosti vysvetľujeme evolučné algoritmy a ich využitie spolu s evolučným obvodom v aplikácii použitej pri testovaní.

4.1 Evolučné a genetické algoritmy

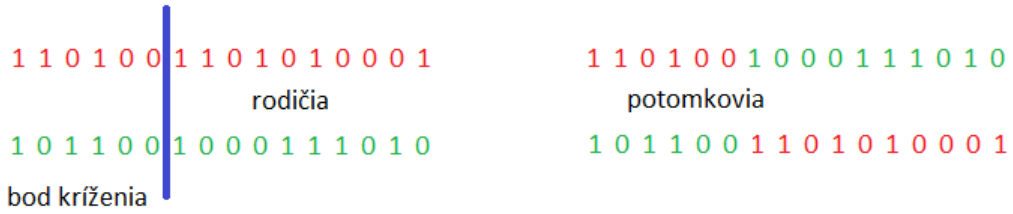
Evolučné algoritmy (Evolutionary algorithms) sú algoritmy, ktoré využívajú spôsoby založené na genetike a evolúcii, zvyčajne na riešenie zložitých matematických a iných úloh. Existuje viacero podtried evolučných algoritmov, ktoré sa líšia v závislosti na riešenom probléme a implementačných detailoch. V našom prípade ide o tzv. Genetické algoritmy (Genetic algorithms), kde riešený problém a jeho dáta reprezentujeme vo forme znakových reťazcov, väčšinou v binárnej forme.

Genetický algoritmus počas svojho behu pracuje s populáciou reťazcov (reprezentuje jedno z možných riešení zadaného problému), ktorá obsahuje množstvo individuálnych jedincov. Evolúcia na začiatku pracuje s náhodne vygenerovanými jedincami, pri ktorých nás zaujíma ich fitness (hodnota, ktorá reprezentuje úspešnosť daného jedinca). Na základe hodnôt fitness vyberáme vhodných jedincov, modifikujeme pomocou rôznych techník ako sú napríklad kríženie (crossover) a mutácia (mutation) so snahou priblížiť sa správnemu riešeniu. Takto vzniká nová populácia, ktorá bude následne použitá v ďalšej iterácii algoritmu. Beh algoritmu je ukončený v prípade, že bolo vygenerované maximálne množstvo generácií alebo bola dosiahnutá uspokojivá hodnota fitness. Použité pojmy vysvetľujeme nasledovne:

- Kríženie (crossover) - proces kríženia vyžaduje minimálne 2 jedincov, označovaných tiež ako rodičov (parents), z ktorých následne vznikajú potomkovia (children). Kríženie si zvolí spoločný bod u obidvoch rodičov, následne všetky dáta za zvoleným bodom zamení, čím vznikajú potomkovia.

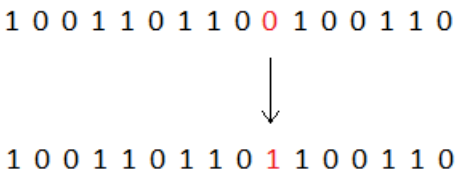
  Ilustrovaný príklad zobrazuje tzv. jednobodové kríženie. Existuje množstvo ďalších spôsobov, ako napríklad dvojbodové kríženie (u rodičov volíme 2 body, ktoré ohraničujú zamenené dáta), alebo spôsob, kde pre každého rodiča volíme bod kríženia osobitne, čím vznikajú potomkovia rozličnej dĺžky.

- Mutácia (mutation) - mutácia v genetických algoritmoch poskytuje možnosť zmeny



Obrázok 4.1: Proces kríženia jedincov

bitov v reťazci na hodnotu opačnú ako bola pôvodná. Pravdepodobnosť zmeny každého z bitov býva zvyčajne 1/n, kde n je dĺžka bitovej postupnosti.



Obrázok 4.2: Proces mutácie u jedinca

- Fitness - funkcia fitness je spôsob ohodnotenia výstupu genetického algoritmu. Jej úlohou je určiť, ako sa daná generácia približuje skutočnému riešeniu, na základe čoho následne vyberáme vhodných jedincov, z ktorých formujeme ďalšiu generáciu algoritmu.

4.2 Evolučné obvody, Sensor Security Simulator

Aplikácia Sensor Security Simulator [12] bola pôvodne vyvinutá Masarykovou Univerzitou na riešenie vybraných problémov bezpečnosti IT. Kód je napísaný v jazyku C++ pre operačné systém MS Windows 2000/XP/Vista. Jednou z implementovaných funkcionalít je simulácia evolučného obvodu, založeného na princípe evolučného algoritmu s použitím knižnice GAlib [6].

Obvod tvorí vstupná vrstva, výstupná vrstva a niekoľko interných vrstiev. Vrstvy sú medzi sebou prepojené pomocou konektorov, ktoré ovplyvňujú veľkosť vstupu z predchádzajúcej vrstvy. Uzly v obvode predstavujú funkcie, ktoré môžu byť nasledovné:

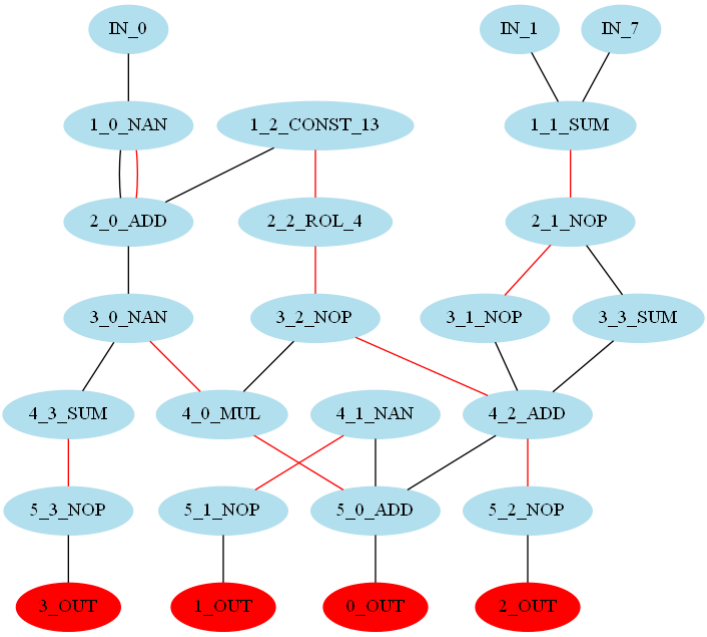
- FNC_NOP - žiadna operácia, funkcia predá na výstup rovnakú hodnotu ako získala.
- FNC_OR - vykoná sa operácia disjunkcie na vstupoch

- FNC_AND - vykoná sa operácia konjunkcie na vstupoch
- FNC_CONST - na výstup vraciame konštantu
- FNC_XOR - vykoná na vstupoch operáciu exkluzívnej disjunkcie
- FNC_NOR - vykoná na vstupoch negáciu logického súčtu (disjunkcie)
- FNC_NAND - vykoná na vstupoch negáciu logického súčinu (konjunkcie)
- FNC_ROTL - vykoná na vstupoch operáciu bitového posunu vľavo
- FNC_ROTR - vykoná na vstupoch operáciu bitového posunu vpravo
- FNC_BITSELECTOR - vykoná na vstupoch operáciu bitového súčinu
- FNC_SUM - vykoná na vstupoch operáciu súčtu
- FNC_SUBS - vykoná na vstupoch operáciu rozdielu
- FNC_ADD - vykoná na vstupoch operáciu súčtu, navyše oproti funkcii FNC_SUM pripočíta hodnotu uzlu na rovnakej pozícii z prechádzajúcej vrstvy
- FNC_MULT - vykoná na vstupoch operáciu násobenia
- FNC_DIV - funkcia celočíselne delí hodnotu uzlu na rovnakej pozícii z prechádzajúcej vrstvy hodnotami pripojených uzlov
- FNC_READ - funkcia podľa hodnoty uzlu na rovnakej pozícii z prechádzajúcej vrstvy prečíta jeden zo vstupov obvodu

Evolúcia ako bola uvedená v [4.1] sa dostáva na úroveň obvodov, čím zachovávame všetky techniky využívané evolúciou. Tie fungujú nasledovne:

- Kríženie - v pozícii jedincov (rodičov, potomkov) figurujú celé obvody. Proces prebieha na úrovni vrstiev, čo v tomto prípade značí ako vrstvy s funkciami, tak vrstvy s prepojením. Po zvolení bodu kríženia (výber jednej z vrstiev) sú všetky nasledujúce vymenené medzi rodičmi.
- Mutácia - prebieha na úrovni prepojení aj na úrovni funkcií vo vrstve (funkciu nahradíme inou, náhodne vybranou, pridáme/odoberieme prepojenie).
- Fitness - ako bolo spomenuté pri evolučných algoritmoch, úlohou funkcie fitness je ohodnotiť úspešnosť poskytnutého (možného) riešenia. Riešenie úlohy v našom prípade predstavuje obvod.

Ako príklad uvádzame jednoduchý obvod s piatimi vrstvami a počtom uzlov na vrstvu 4. Skutočne však algoritmus generoval 16 vstupov pre obvod, z ktorých sú v tomto stave obvodu zapojené len 3 (vstup 0, 1 a 7). Hodnota každého z uzlov je výsledkom funkcie uvedenej v grafe, ktorá je interne reprezentovaná hodnotou 0-255. V závislosti na funkcii je potom možnosť privádzať uzlu na vstup jednu/niekoľko hodnôt z predchádzajúcej vrstvy.



Obrázok 4.3: Príklad evolučného obvodu

4.3 Náhodné dáta

Aplikácia primárne využíva náhodný generátor postavený na báze funkcie MD5. Výhodou je v tomto prípade spolu s používaním tzv. seedu [1] fakt, že vďaka znalosti seedu sme schopní každé náhodné rozhodnutie algoritmu reprodukovať a tak spustiť vybrané testy znovu.

Uvedený generátor vďaka svojej závislosti na seede a tiež vďaka faktu, že je postavený práve na hashovacej funkcii nieje v niektorých prípadoch využiteľný. Jedná sa napr. o nutnosť použitia skutočne náhodných dát, ktoré dodávame ako vstup SHA-3 kandidátom. Pre tento účel využívame dáta generované kvantovým generátorom dostupným na [11]. Aplikácia má prístup ku 100 súborom o celkovej veľkosti 1,5GB binárnych dát.

1. Náhodné číslo, generované na začiatku každého behu aplikácie. Po vygenerovaní pomocou neho inicializujeme náhodný generátor aplikácie.

4.4 Implementácia kandidátnych funkcií

Pred samotným spúšťaním testov bolo nutné pôvodnú aplikáciu Sensor Security Simulator upraviť, aby bolo neskôr možné spúšťať testy pomocou nástroja BOINC na počítačoch v LaBAK-u [2]. Hlavná funkcionalita bola oddelená od GUI spolu s úpravami kódu, ktoré boli vykonané z dôvodu prípravy aplikácie na platformovú nezávislosť, kedže ju bolo možné používať len na systémoch MS Windows.

Súťaž o novú hashovaciu funkciu bola vypísaná 2. novembra 2007 Americkým štandardizačným inštitútom [3], na konci ktorej výherný algoritmus ponesie pomenovanie SHA-3. Prihlásených algoritmov bolo spolu 64, z ktorých 51 kandidátov postúpilo do prvého kola a 14 ich bolo vybraných do druhého kola. Kandidátne funkcie boli prístupné kryptografickej komunite a na základe spätnej väzby bolo zvolených 5 finalistov. Podmienky na zapojenie sa zahŕňali napríklad použitie jazyka C (podľa normy ANSI C) a tiež implementáciu predpísaného API [13]. Rozhranie popisuje dátové typy a štruktúry používané pri ukladaní potrebných hodnôt počas hashovania, ako aj 4 funkcie určené na výpočet hashu. Podľa rozhrania tiež musí každá z kandidátnych funkcií podporovať výstupné hashe o dĺžke 224, 256, 384 a 512 bitov, poprípade aj iné, ktoré už niesú nutné.

Do testovacej aplikácie sme implementovali všetky algoritmy od 1. kola (s výnimkou dvoch, ktoré upresňujeme v podkapitole [4.5]), čo značí 51 funkcií. Použité boli najnovšie dostupné verzie, tzn. verzie pre posledné kolo, ktorého sa daný algoritmus zúčastnil. V kóde sú používané pomocou spoločného rozhrania (mierne modifikované pôvodné rozhranie), čo znamenalo nutnosť pre každú z nich vytvoriť triedu, ktorá rozhranie implementuje. Funkciu si užívateľ vyberie zmenou hodnoty v konfiguračnom súbore.

4.5 Priebeh testov

Ako sme uviedli v [4.4], počet algoritmov v prvom výberovom kole bol 51. Počas ich implementácie sme vynechali 2, z dôvodu veľkosti zdrojových súborov a rýchlosti (tie presahovali v jednom z prípadov 100 MB). Následne sme pre testovanie vybrali algoritmy, kde bolo možné obmedziť počet rund, keďže testy primárne prebiehali na oslabených variantách, čo vo výsledku znamenalo 34 algoritmov upravených na testovanie. Nastavenie zvoleného počtu rund prebieha taktiež z konfiguračného súboru. Využívali sme funkcie s výstupom dĺžky 256 bitov.

Testovanie prebiehalo simultánne na 15 strojoch Mefitas pomocou aplikácie BOINC určenej na distribuované výpočty. Keďže počet aktuálne spracovaných úloh pre jednotlivé počítače je prideľovaný pre každé jadro procesoru, bolo možné pracovať (podľa aktuálnej dostupnosti) maximálne na 30-tich výsledkoch naraz. Priebeh každého z testov vyzerá nasledovne:

- Vytvárame populáciu niekoľkých obvodov, inicializujeme a nastavujeme podľa hod-

2. Laboratoř bezpečnosti a aplikované kryptografie.
3. National Institute of Standards and Technology.

nôt v konfiguračnom súbore, ustanovujeme (popr. načítame) seed pre náhodné generátory, generujeme prvé testovacie vektory.

- Evaluácia - ohodnotenie výsledkov, ktoré nám obvody vrátia.
- Evolúcia - využívame techniky evolúcie (mutácia, kríženie), formujeme obvody nové. Následne opakujeme hodnotenie a porovnávame výsledky s predchádzajúcimi (sú výsledky lepšie?). Tento krok spolu s minulým opakujeme pre každú generáciu, pričom v istých intervaloch generujeme aj nové testovacie vektory.
- Finálne hodnotenie - na záver dostáva obvod nové dáta, na ktorých sa predtým nikdy neučil. Vyberáme najlepšieho jedinca, ktorého hodnotíme. Túto výslednú hodnotu používame pri prezentácii výsledkov testov.

Okrem spomínanej finálnej hodnoty fitness program generuje niekoľko priebežných výstupov v rôznych formátoch, ktoré nás taktiež zaujímajú:

- EAC_fitnessProgress.txt - každých niekoľko (napr. 10) generácií ukladáme hodnoty ako sú maximálna a priemerná fitness, celkové maximum dosiahnuté počas behu. Maximálne a priemerné hodnoty fitness ukladáme tiež v odobitných súboroch, ktoré je možné použiť na generovanie grafov a pod.
- EAC_circuit.dot - formát dot je používaný pri generovaní grafov obvodu, vytvárané sú tiež samostatné súbory pre každé nové nájdené maximum hodnoty fitness.
- EAC_circuit.txt - textová reprezentácia obvodu, generujeme podobne ako u .dot formátu.
- EAC_circuit.bin - súbor s uloženým obvodom v binárnej forme, využívaný v prípade, že chceme pokračovať s už získaným obvodom.
- EAC_circuit.c - zdrojový súbor obvodu, ktorý je možné skompilovať.

Kapitola 5

Spôsoby testovania a výsledky

5.1 Rozlišovanie pri obmedzenom vstupe funkcie

Úlohou obvodu bolo správne odhadnúť pôvod dodaných dát a tak správne rozlišovať medzi dvomi rôznymi typmi vstupných dát. Hashovacej funkcii dodávame vstup so známou štruktúrou.

5.1.1 Testovacie vektory

Generovanie sady testovacích vektorov začína výberom a prípravou náhodných dát z kvantového generátoru [4.3]. Každú sadu tvorí niekoľko sto až tisíc vektorov, v závislosti na nastavení, pričom pri generovaní novej sady je znovu náhodne zvolený jeden zo súborov s dátami. Následne prebieha výber, ktorý z 2 typov vektorov budeme generovať. Presná postupnosť generovania nieje nijak určená, ale algoritmus po vygenerovaní celej sady zaručuje vyvážený pomer medzi vektormi. Tieto vektory sú:

- Náhodné dáta – pseudo-náhodné dáta, načítané priamo z vybraného súboru. Dáta boli načítavané po blokoch, ktorých dĺžka závisí na dĺžke výstupných hashov.

- Hash – výstupy hashovacej funkcie. Vstupy funkcie tvorili bloky rovnakej dĺžky ako výstupný hash, nie však ľubovoľné. Tvorili sme bloky vo formáte: `XYXYXYXY.....XY`, tzn. bloky, ktoré tvorili 2 opakujúce sa byty.

Každý testovací vektor počas jeho generovania označujeme, či sa jedná o náhodné dáta (označené hodnotou 0xff hexadecimálne), alebo o výstup hashovacej funkcie (označené hodnotou 0x00). Výsledná sada môže následne vyzerať ako:

```
1.          <473029411faa6733015ed10158d115c9; 0x00>
2.          <98d603fa3425c6ed06be541806ce22c1; 0x00>
3.          <dd82d62b9acc87f233ade01e233b6857, 0xff>
4.          <bb8398b1baca81887d5d8ed04f9a24d9, 0x00>
.....
.....
200.        <bddd61390db6b1658e879c816d0dbd9c, 0xff>
```

5.1.2 Vstupy obvodu

Ako vstup poskytujeme evolučnému obvodu prvú časť vektoru, tzn. náhodné dáta, alebo hash. Z toho dôvodu sa veľkosť vstupnej vrstvy obvodu nastavuje podľa dĺžky týchto dát.

5.1.3 Výstupy obvodu

Očakávaným výstupom obvodu je určenie, či na vstupe boli náhodné dáta, alebo hash. Na tento účel postačuje výstupná vrstva tvorená jediným uzlom.

5.1.4 Spôsoby predikcie

Ako sme uviedli, očakávané výstupy obvodu sú v tomto prípade 2 možné hodnoty, z ktorých jedna reprezentuje hash, druhá náhodné dáta. Aj prípade nastavenia výstupnej vrstvy obvodu na minimum, tzn. 1 uzol (reprezentovaný premennou typu unsigned char) dostávame výstupné hodnoty v rozsahu 0-255, ktoré musíme nejakým spôsobom porovnať so „správnym“ výstupom a určiť mieru zhody. Naskytli sa nám tak 2 možné prediktory:

- Predikcia na základe číselnej hodnoty - výstupy v rozsahu 0-127 sú považované za hash, na druhej strane tie v rozsahu 128-255 sú považované za náhodné dáta.

- Predikcia na základe hammingovej váhy – hodnoty s váhou 0-4 reprezentovali hash, s váhou 5-8 reprezentovali náhodné dáta.

Prvými testmi, realizovanými na funkcii MD5 uvedených v [5.1.7] sa ukázal lepší prvý spôsob predikcie. Počas predikcie počítame počet správne predikovaných hodnôt (správne určenie o aký vstup sa jednalo) a celkový počet predikcií. Hodnotu fitness nám následne určuje vzťah `fit = predikovane_spravne / predikovane_celkovo`.

Hodnota funkcie fitness je v tomto prípade určená podielom správne odhadnutých výstupov ku celkovému počtu vykonaných odhadov (1*200, tzn. jedna odhadovaná hodnota na každý z 200 testovacích vektorov v sade).

5.1.5 Rozšírenie vstupov obvodu

Ďalšou z možností testovania bolo poskytnúť obvodu viac dát na rozhodovanie, napríklad 256 bytov na rozdiel od pôvodných X bytov (pre SHA-3 32 bytov). Skutočná veľkosť vstupnej vrstvy obvodu sa však nemení, preto má obvod na začiatku dostupných pôvodných X bytov. Plnú dĺžku využívame prostredníctvom funkcie FNC_READ [4.2], ktorá podľa vstupu z predchádzajúcej vrstvy obvodu v rozsahu 0-255 načíta hodnotu vstupnej vrstvy na danej pozícii ako zobrazuje Obrázok A.1.

Samotné testovacie vektory, konkrétne hodnoty vstupnej vrstvy obvodu sú z dôvodu pevnej dĺžky výstupu hashovacej funkcie tvorené zreťazením niekoľkých hashov Obrázok A.2. V prípade náhodných dát postupujeme rovnakým spôsobom.

5.1.6 Nastavenie evolúcie a obvodu

Veľkosť populácie	20
Počet testovacích vektorov	200
Pravdepodobnosť mutácie	0,05
Pravdepodobnosť kríženia	0,5
Počet generácií evolúcie	100000
Počet vrstiev v obvode	8
Veľkosť vstupnej vrstvy obvodu	32
Veľkosť vnútorných vrstiev obvodu	16
Veľkosť výstupnej vrstvy obvodu	1
Počet konektorov na vrstvu	16
Frekvencia zmeny testovacích vektorov	po 10 generáciách [a]

a. Obvod potrebuje istý čas na učenie sa, no na druhej strane je schopný z jednej sady testovacích vektorov získať len obmedzené znalosti. Najlepšie sa po prvých testoch na MD5 prejavila verzia s 10-generačnými zmenami.

Tabuľka 5.1: Nastavenie evolúcie a obvodu

5.1.7 Testy na funkcii MD5

Prvé testy vykonané na funkcii MD5 nám pomohli rozhodnúť pri niektorých z nastavení evolučného obvodu, ako napríklad výber prediktoru pri rozlišovaní, frekvenciu zmeny testovacích vektorov a pod. Ako prvý uvádzame rozdiel medzi prediktormi. Vychádzali sme z porovnania 4 a 5 rundovej verzie pri zmene testovacích vektorov po 10 generáciách a výslednej hodnoty pre 15 behov každej z nich (výsledky dosiahnuté po 100000 generáciách).

	4 rundy	5 rund
Prediktor č. 1	0,879	0,6663
Prediktor č. 2	0,826	0,6177

Tabuľka 5.2: Nastavenie evolúcie a obvodu

Jeden z úspešných behov 4-rundovej verzie pri zmene testovacích vektorov (skrátene TV) po 10 generáciách a použití prediktoru č.1 vykresľujeme Obrázok A.3 a tiež pre prediktor č.2, kde sa obvod učí zhruba po rovnakom čase, priemerne ale nedosahuje výsledky ako prvá verzia predikcie Obrázok A.4. Z grafov vidíme, že krivka hodnôt fitness nemá čisto rastúcu tendenciu. Toto je spôsobené hlavne zmenou TV v pravidelných intervaloch, kedy fitness poklesne.

Následne sme testovali jednotlivé kombinácie nastavení počtu rund a zmeny TV. Hodnoty v tabuľkách značia priemernú hodnotu výslednej fitness všetkých 10 behov testu. Výsledky pre rundy 1-3 neuvádzame, MD5 pri tomto nastavení generuje istú časť výstupného hashu konštantnú, obvod ju preto rozpozná behom niekoľkých generácií. Výsledky poslúžili k faktu, že sme pri testovaní SHA-3 kandidátov zvolili zmenu TV každých 10 generácií.

	1	10	100
4 rundy (200000 gen.)	0,8	0,99	0,9475
5 rund (200000 gen.)	0,5	0,83	0,6525
6 rund (200000 gen.)	0,51	0,4917	0,4883
7 rund (200000 gen.)	0,5	0,52	0,4883

Tabuľka 5.3: Nastavenie evolúcie a obvodu

Posledným z testov bolo zistenie vplyvu väčších vstupných dát na priebeh evolúcie, ako sme uvádzali v [5.1.5]. Nasleduje porovnanie výsledkov pre 4 a 5 rundovú verziu so zmenou po 10 a 100 generáciách. Verzia s pôvodných 16B vstupom nepoužíva funkciu READ:

	10	100
4 rundy, 16B	0,99	0,975
4 rundy, 256B	0,842	0,992
5 rund, 16B	0,83	0,6525
5 rund, 256B	0,6815	0,567

Tabuľka 5.4: Nastavenie evolúcie a obvodu

Z výsledkov pre 4 rundovú verziu sa pre 256 vstupov javí vhodné meniť TV menej frekventovane (po 100 generáciách), než u prvej varianty. Po zvýšení rund na 5 však nedostávame lepšie výsledky, práve naopak, varianta sa prejavila horšie ako pri vstupe dĺžkou 16 bytov, preto sme tento spôsob d'alej nepoužívali.

5.1.8 Výsledky pre SHA-3 kandidátov

Zvolený počet rund u funkcií uvádzame pri každej z nich v zátvorke. Niektoré z funkcií nemalo zmysel testovať s počtom rund menším, než sme zvolili a to z dôvodu konštantnej časti výstupu, popr. celého výstupu. Uvedená hodnota určuje priemer finálnych hodnôt fitness [4.5] z 10tich behov testu. Z výsledkov je zjavné, že vo väčšine prípadov obvod žiadne závislosti vo vstupných hashoch neodhalil, preto sa výsledky pohybujú okolo hodnoty 0,5, tzn. aj na konci evolúcie rozlišujeme medzi hashom a náhodnými dátami s 50% úspechom.

Abacus(1)	Arirang(4)	Aurora(1)	Blake(1)	Blender(1)	BMW [a](1)	Boole(1)
0,5195	0,4935	0,506	0,4935	0,503	0,4895	0,4965

a. Blue Midnight Wish

Cheetah(2)	CHI(1)	Crunch(34)	Cubehash(1)	DCH(1)	DSHA(5) [a]	DSHA 2(1)
0,4845	0,5045	0,516	0,4905	0,5145	0,7815	0,5135

a. Dynamic SHA

ECHO(1)	ESSENCE(1)	Fugue(1)	Grostl(1)	Hamsi(1)	JH(7)	Lesamnta(3)
0,497	0,481	0,496	0,529	0,502	0,563	0,606

Luffa(8)	Lux(1)	MD6(7)	MeshHash(1)	SANDstorm(3)	Sarmal(1)	SHAvite3(2)
0,531	0,498	0,4975	0,501	0,5325	0,5375	0,4865

SIMD(1)	Tangle(7)	Tib3(1)	Twister(7)	WaMM(2)	Waterfall(1)
0,482	0,4945	0,5335	0,504	0,4985	0,5085

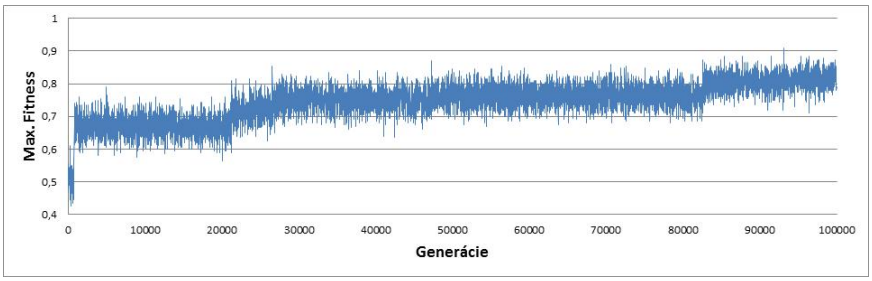
Tabuľka 5.5: Výsledky pre SHA-3 kandidátov

Jediný úspech zaznamenal náš postup v prípade funkcie Dynamic SHA, kde správne rozlišoval hash od náhodných dát v priemere s úspešnosťou 78%. V testoch sme preto pokračovali s postupným zvyšovaním počtu rund na 8 (zo 16) a tiež zvýšeným počtom generácií na 200000. Nasledujúca tabuľka zobrazuje výsledky z jednotlivých behov pre každé nastavenie:

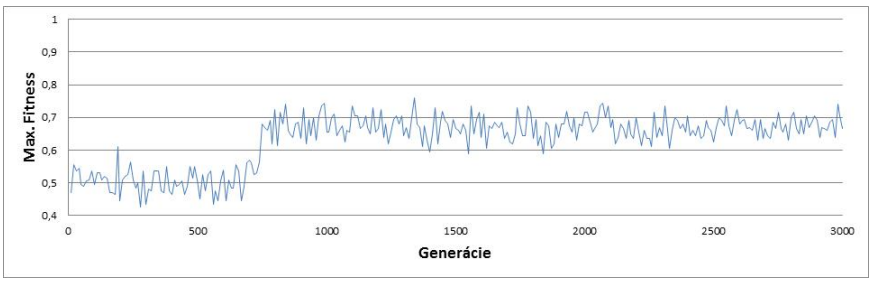
	5 rund	6 rund	7 rund	8 rund
1.	0,755	0,78	0,895	0,535
2.	0,79	0,87	0,74	0,55
3.	0,79	0,8	0,845	0,5
4.	0,755	0,805	0,825	0,5
5.	0,765	0,75	0,795	0,515
6.	0,855	0,815	0,73	0,47
7.	0,785	0,75	0,785	0,505
8.	0,785	0,79	0,725	0,52
9.	0,765	0,79	0,785	0,535
10.	0,77	0,75	0,74	0,52

Tabuľka 5.6: Testy na Dynamic SHA

Pre verziu s 5 rundami uvádzame behy číslo 2 a neskôr aj 3. Hodnoty grafu zobrazujú maximálne hodnoty fitness namerané pri hodnotení, ktoré sme opakovali každých 10 generácií. Prvé skokové vylepšenie evolúcie pozorujeme už po 750 generáciách, pričom neskôr nasledujú ďalšie 2. V druhom grafe vidíme detail na prvých 3000 generácií behu.

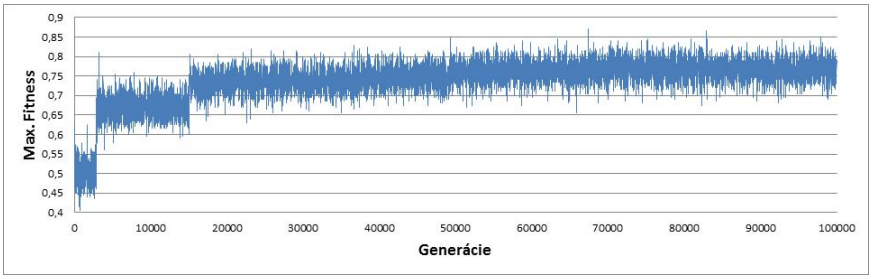


Obrázok 5.1: 2. beh pre 5 rundovú verziu



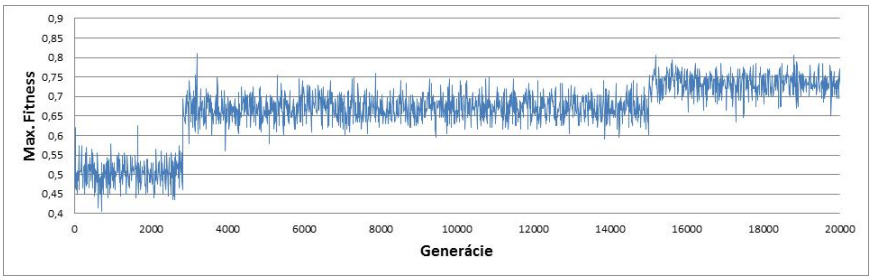
Obrázok 5.2: Detail 2. behu pre 5 rundovú verziu

Následne pre porovnanie uvádzame nájdené obvody počas behu. Prvý z nich je obvod pre fitness 0,53 Obrázok A.5, ktorý odhaduje vstupy len s priemerným úspechom. Obvod je z z rannej fázy evolúcie, keďže využíva len niekoľko z prvých vrstiev. Druhý obvod Obrázok A.6 je už lepší, pričom využíva viac vstupov než predchádzajúca verzia. Finálna verzia obvodu potom vyzerá nasledovne Obrázok A.7. Beh číslo 3, ktorý skončil s rovnakým výsledkom uvádzame ako druhý príklad. Zo začiatku evolúcie prebieha podobne ako v predchádzajúcom behu, s rýchlym skokom, neskôr však pozorujeme pomalý nárast evolúcie (od generácie 15000 do 60000).



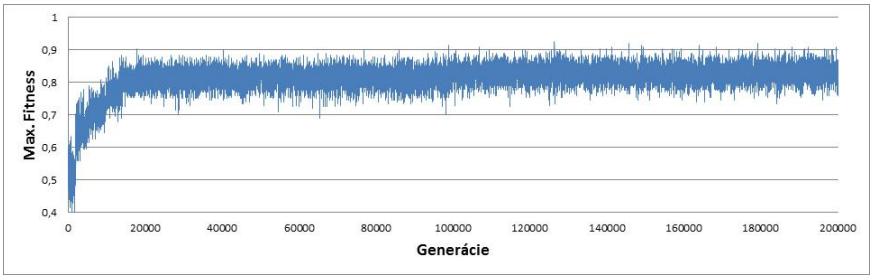
Obrázok 5.3: Detail 3. behu pre 5 rundovú verziu

Nasledujúce grafy zobrazujú najlepšie behy pre 6 a 7 rundovú verziu funkcie Dynamic SHA. Pre lepšie porovnanie s predchádzajúcou verziou je uvedený tiež graf pre prvú polo-

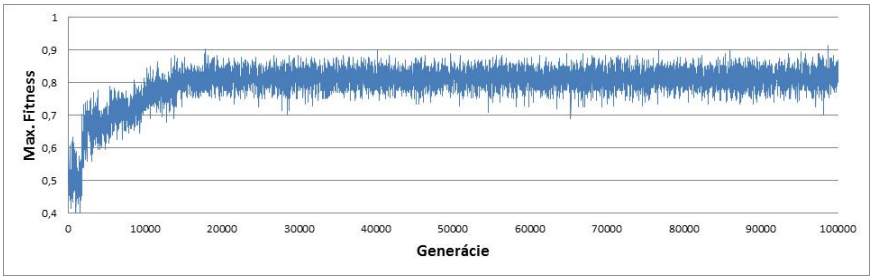


Obrázok 5.4: Detail 3. behu pre 5 rundovú verziu

vicu behu. Evolúcia prebieha vo všetkých troch prípadoch podobne, s rýchlym rastom na začiatku, obvodu sa však ani pri zvýšení počtu generácií na dvojnásobok nepodarilo naučiť viac

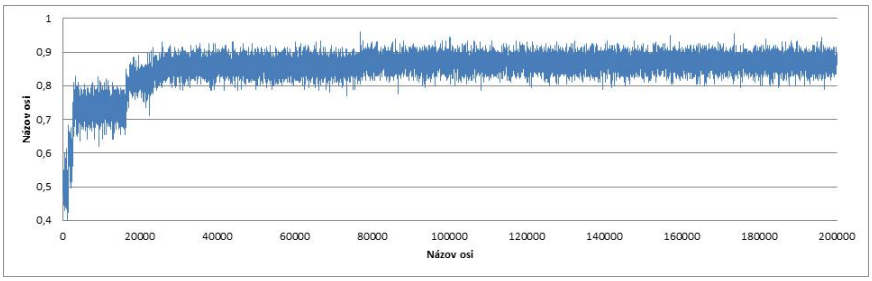


Obrázok 5.5: 2. beh pre 6 rundovú verziu

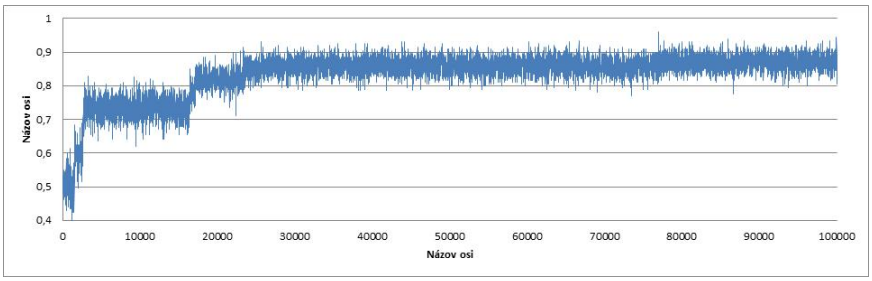


Obrázok 5.6: Prvých 100000 generácií behu pre 6 rundovú verziu

Obvody pre uvedené 6 Obrázok A.8 a 7 Obrázok A.8 rundové behy sú podobné, než u 5 rundovej verzie. Môžeme si napr. všimnúť, že na výsledok obvodu nemajú skoro žiadny vplyv vstupy 0-12, z ktorých väčsina nieje do obvodu zapojená.



Obrázok 5.7: 2. beh pre 7 rundovú verziu



Obrázok 5.8: Prvých 100000 generácií behu pre 7 rundovú verziu

5.2 Odhad obmedzenej časti vstupných dát

Úlohou obvodu je zo znalosti hashov odhadovať informácie o malej časti vstupného bloku.

5.2.1 Testovacie vektory

Sada testovacích vektorov je na rozdiel od minulého prípadu tvorená jedným typom dát. Prvou časťou vektoru je plaintext, pričom ho tvoria dáta načítané z kvantového generátoru [4.3]. Takto načítaný blok náhodných dát je hashovaný vybranou funkciou, ktorej výstup tvorí druhú časť testovacieho vektoru.

5.2.2 Vstupy obvodu

Na vstup obvodu privádzame výsledné hashe. Rovnako ako v predchádzajúcich testoch používame 256 bitovú dĺžku hashov.

5.2.3 Výstupy obvodu

Očakávaným výstupom je odhad informácií o malej časti vstupu hashovacej funkcie, ktorý daný hash vyprodukoval. Používali sme vrstvu o veľkosti 2 uzlov a teda odhadovali informácie o prvých dvoch bytoch vstupného bloku funkcie.

5.2.4 Spôsob predikcie

Spôsob predikcie je založený na porovnávaní hammingovej váhy a to nasledovne:

- Výpočet hammingovej váhy pre aktuálne porovnávané byty.
- Pred porovnaním výsledkov vypočítame zhodu na body a to nasledovným spôsobom: `points = NUM_BITS - abs(predictWeight - correctWeight)` (konštanta NUM_BITS značí počet bitov na 1 byte, teda 8, predictWeight a correctWeight ukladajú hammingovu váhu). Takto vypočítaná hodnota poskytuje obvodu presnejšie informácie o zlepšení/zhoršení evolúcie, keďže obvod nedostáva len informáciu, či hammingovu váhu trafil, ale tiež vie, nakoľko sa k zhode v prípade neúspechu priblížil. Hodnotu „points“ následne delíme počtom vsetkých predikovaných bytov a dostávame výstupnú hodnotu fitness.

5.2.5 Nastavenie evolúcie a obvodu

Veľkosť populácie	20
Počet testovacích vektorov	200
Pravdepodobnosť mutácie	0,05
Pravdepodobnosť kríženia	0,5
Počet generácií evolúcie	100000
Počet vrstiev v obvode	8
Veľkosť vstupnej vrstvy obvodu	32
Veľkosť vnútorných vrstiev obvodu	16
Veľkosť výstupnej vrstvy obvodu	2
Počet konektorov na vrstvu	16
Frekvencia zmeny testovacích vektorov	po 10 generáciách

Tabuľka 5.7: Nastavenie evolúcie a obvodu

5.2.6 Výsledky pre SHA-3 kandidátov

Počet rund sme zvolili rovnako ako pri predchádzajúcich testoch. Uvedená hodnota určuje priemer finálnych hodnôt fitness [4.5] z 10tich behov testu, pričom reprezentuje mieru zhody ako sme uviedli v XrefId[??]. Výsledky sa pohybujú priemerne okolo hodnoty 3,5; čo značí v priemere 3 až 4 správne odhadnuté bity na jeden byte. Hodnota je nižšia ako priemer, čo značí, že sa obvod nebol schopný pri daných podmienkach učiť a nezískal znalosti pre lepší odhad vstupov.

Abacus(1)	Arirang(4)	Aurora(1)	Blake(1)	Blender(1)	BMW [a](1)	Boole(1)
3,58975	3,587875	3,600875	3,5945	3,5945	3,598375	3,60525

a. Blue Midnight Wish

Cheetah(2)	CHI(1)	Crunch(34)	Cubehash(1)	DCH(1)	DSHA(5) [a]	DSHA 2(1)
3,599625	3,60825	3,592	3,57225	3,599625	3,579875	3,584625

a. Dynamic SHA

ECHO(1)	ESSENCE(1)	Fugue(1)	Grostl(1)	Hamsi(1)	JH(7)	Lesamnta(3)
3,594625	3,5645	3,5915	3,596125	3,58025	3,578125	3,60625

Luffa(8)	Lux(1)	MD6(7)	MeshHash(1)	SANDstorm(3)	Sarmal(1)	SHAvite3(2)
3,609125	3,57828125	3,5959375	3,5996875	3,59109375	3,58625	3,58265625

SIMD(1)	Tangle(7)	Tib3(1)	Twister(7)	WaMM(2)	Waterfall(1)
3,59265625	3,5940625	3,5765625	3,61921875	3,6	3,58640625

Tabuľka 5.8: Výsledky pre SHA-3 kandidátov

5.3 Avalanche efekt

Úlohou obvodu je hľadať a porovnávať vstupné bloky, pre ktoré bude vo výsledných hodnotách hashov rozdielnych viac, resp. menej bitov, než je očakávaná hodnota 1/2.

5.3.1 Testovacie vektory

Testovacie vektory sú v tomto prípade tvorené 16 bytovými blokmi náhodných dát. Dáta využívame ako vstup pre evolučný obvod, tiež ich ale potrebujeme vo fáze predikcie.

5.3.2 Vstupy obvodu

Obvodu na vstup dodávame testovacie vektory v podobe blokov náhodných dát. Veľkosť vstupnej vrstvy obvodu je prispôsobená dĺžke týchto dát.

5.3.3 Výstupy obvodu

Očakávaným výstupom obvodu je blok dát rovnakej dĺžky ako pôvodný vstup. Hľadáme však také bloky, pre ktoré sa budú výstupné hodnoty hashov odlišovať vo väčšom/menšom počte bitov, ako je 1/2.

5.3.4 Spôsob predikcie

Spôsob predikcie je založený na porovnávaní bitov medzi výstupnými hashmi testovanej funkcie. Hashovanie dát je však nutné vykonávať až ne úrovni prediktoru, pretože testovacie vektory aj obvod pracujú s dátami nehashovanými. Predikcia prebieha v nasledujúcich krokoch:

- Porovnanie vstupných a výstupných dát obvodu - nutnosť porovnania vychádza z faktu, že algoritmus v krátkom čase nájde ideálny obvod, ktorý bude len vracať nezmenené vstupné dáta na výstup, čím samozrejme vzniká pri porovnaní hashov kolízia na celej dĺžke. Takto vytvoreným obvodom automaticky priraďujeme hodnotu fitness 0.
- Hashovanie vstupu a výstupu obvodu zvolenou kandidátnou funkciou.
- Porovnanie získaných hashov z minulého kroku na celej ich dĺžke. Jedná sa o porovnanie každého z bitov. Hodnotu fitness tak počítame ako `fitness = pocet_zhodnych_bitov / dlzka_hashu` (využitie hashov o dĺžke 256 bitov zostáva).

5.3.5 Nastavenie evolúcie a obvodu

Na rozdiel od minulých testov sme nastavenia mierne upravili, keďže ide o priemerné hodnoty fitness počas celého behu. Snažíme sa preto obvodu dodávať väčšie množstvo rôznych dát (zníženie počtu a častejšie generovanie nových testovacích vektorov).

Veľkosť populácie	20
Počet testovacích vektorov	100
Pravdepodobnosť mutácie	0,05
Pravdepodobnosť kríženia	0,5
Počet generácií evolúcie	100000
Počet vrstiev v obvode	8
Veľkosť vstupnej vrstvy obvodu	16
Veľkosť vnútorných vrstiev obvodu	16
Veľkosť výstupnej vrstvy obvodu	16
Počet konektorov na vrstvu	16
Frekvencia zmeny testovacích vektorov	každú generáciu

Tabuľka 5.9: Nastavenie evolúcie a obvodu

5.3.6 Výsledky pre SHA-3 kandidátov

Počet rund je volený rovnako, ako pri predchádzajúcich testoch. V tomto prípade sme pri priemerovaní výsledkov nepoužívali finálnu hodnotu fitness, ale brali sme do úvahy priemerné hodnoty počas celého behu. Ako ukazujú výsledky, funkcie sa pohybujú okolo pred-

pokladanej hodnoty 0,5; pričom číslo nižšie ako 0,5 znamená väčší počet zmenených bitov v priemere.

Abacus(1)	Arirang(4)	Aurora(1)	Blake(1)	Blender(1)	BMW [a](1)	Boole(1)
0.476563	0.464844	0.527344	0.464844	0.449219	0.46875	0.519531

a. Blue Midnight Wish

Cheetah(2)	CHI(1)	Crunch(34)	Cubehash(1)	DCH(1)	DSHA(5) [a]	DSHA 2(1)
0.480469	0.472656	0.429688	0.503906	0.484375	0.507813	0.488281

a. Dynamic SHA

ECHO(1)	ESSENCE(1)	Fugue(1)	Grostl(1)	Hamsi(1)	JH(7)	Lesamnta(3)
0.46875	0.464844	0.484375	0.511719	0.445313	0.484375	0.507813

Luffa(8)	Lux(1)	MD6(7)	MeshHash(1)	SANDstorm(3)	Sarmal(1)	SHAvite3(2)
0.460938	0.476563	0.46875	0.480469	0.46875	0.472656	0.476563

SIMD(1)	Tangle(7)	Tib3(1)	Twister(7)	WaMM(2)	Waterfall(1)
0.488281	0.488281	0.429688	0.464844	0.476563	0.445313

Tabuľka 5.10: Výsledky pre SHA-3 kandidátov

Kapitola 6

Záver

Cieľom práce bolo niekoľkými spôsobmi otestovať kandidátne hashovacie funkcie SHA-3 so zameraním na vyhľadávanie závislostí vo výstupných hashoch. V teoretickej časti rozoberáme hashovacie funkcie, ich rozdelenie a funkcionalitu. Popisujeme základné typy útokov na jednotlivé druhy funkcií a tiež prakticky vykonané útoky na používaných funkciách s uvedenými príkladmi.

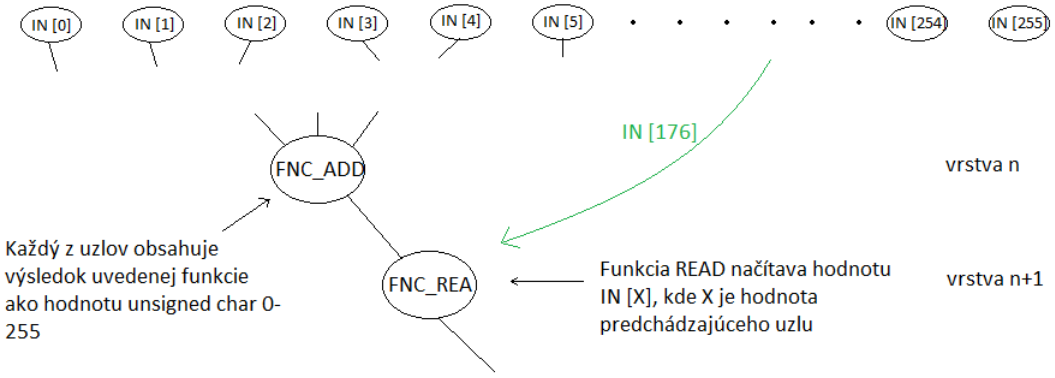
Praktická časť spájala prípravu testovacej aplikácie, implementáciu testovaných funkcií a tiež testy. Použité nastavenia obvodu vychádzali z už známeho chovania a testov, na ktoré bola aplikácia využívaná v minulosti, ale aj z vlastných testov, vykonaných na hashovacej funkcii MD5. Tiež musíme poznamenať, že zvolené nastavenia pravdepodobne nie sú jediné vhodné pre hľadanie riešenia nášho problému.

Výsledky práce sme sústredili na popis vykonaných testov a tiež výsledkov fitness, ktoré nám poskytujú prehľad o odolnosti zapojených kandidátov voči našim útokom. Najlepšie výsledky sme dosiahli pri funkcii DynamicSHA, kde sa prvou metódou testovania odhalila závislosť, aj keď len pri obmedzenom počte rund. Najviac perspektívne sa preto prejavil prvý testovací spôsob so známou štruktúrou vstupnej správy.

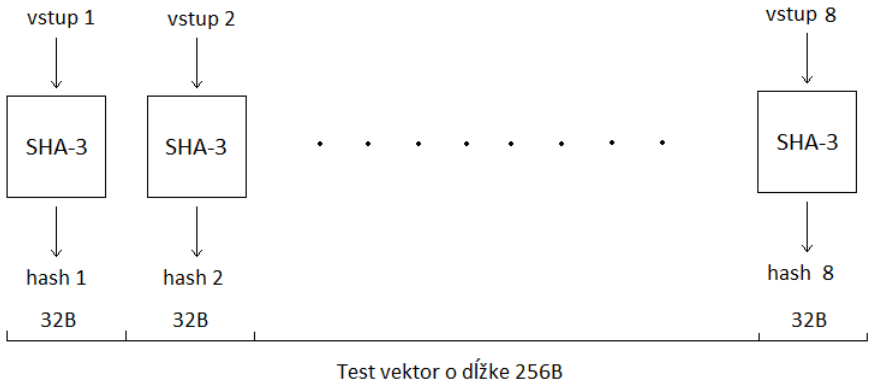
Vďaka tejto práci som nadobudol množstvo nových znalostí, či už sa jedná o znalosti v problematike hashovacích funkcií a útokov na ne, tiež aj genetických algoritmov, s ktorými som sa stretol poprvý raz. Tiež som si uvedomil nutnosť správnej funkcionality a odolnosti hashovacích funkcií voči útokom, ktoré tak poskytujú dostatočnú bezpečnosť vo všetkých oblastich využitia, kde je potrebná.

Dodatok A

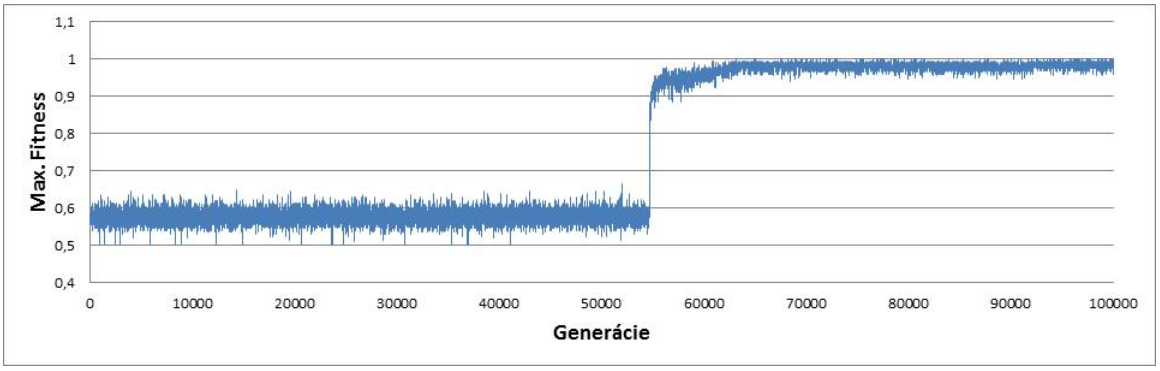
Grafy



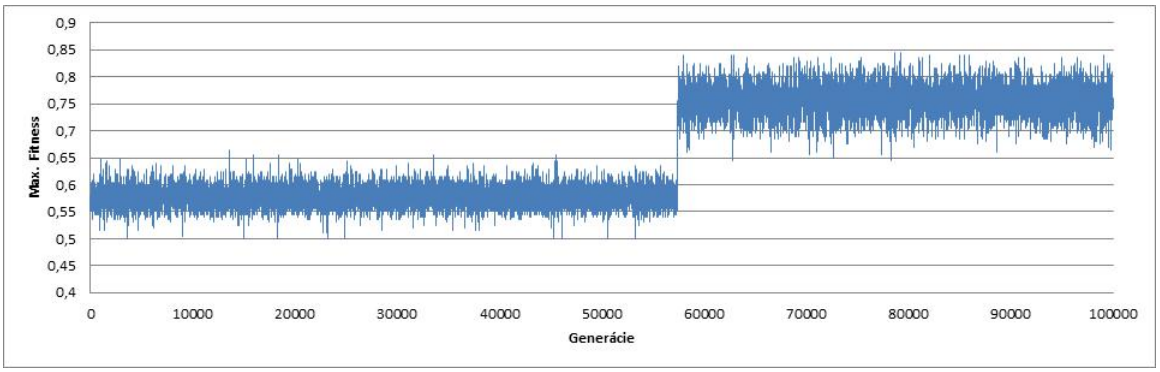
Obrázok A.1: Funkcia FNC_READ



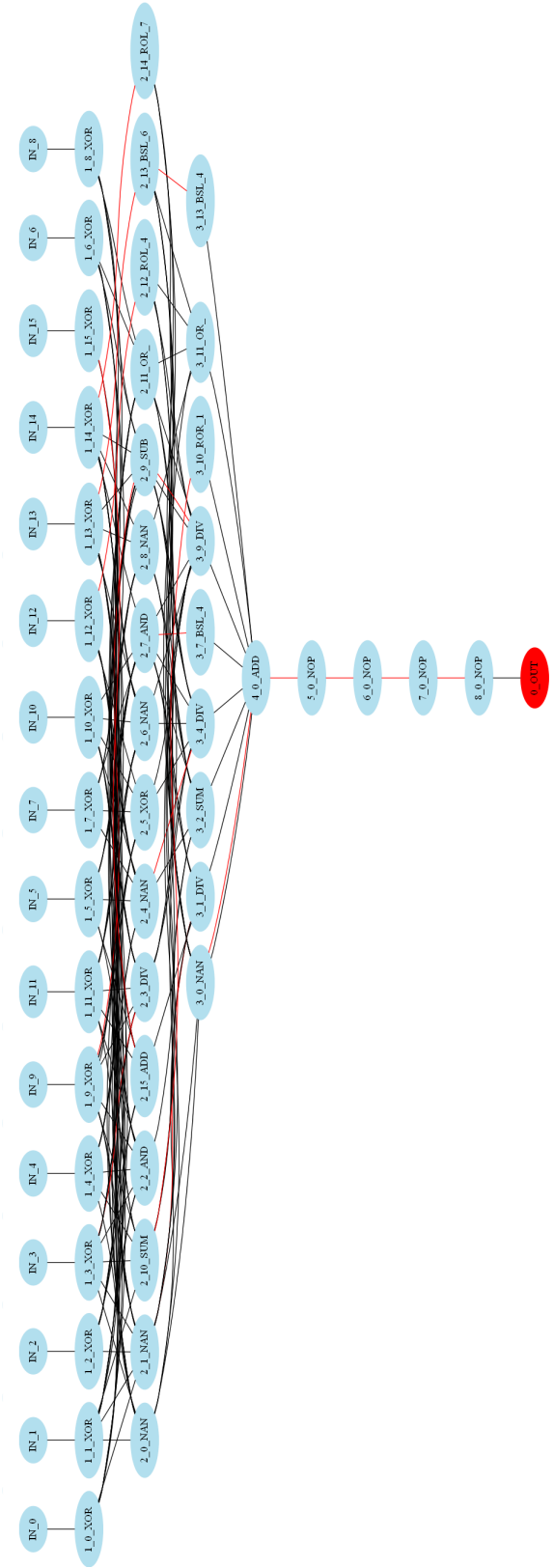
Obrázok A.2: Tvorba testovacieho vektoru o dĺžke 256 bytov



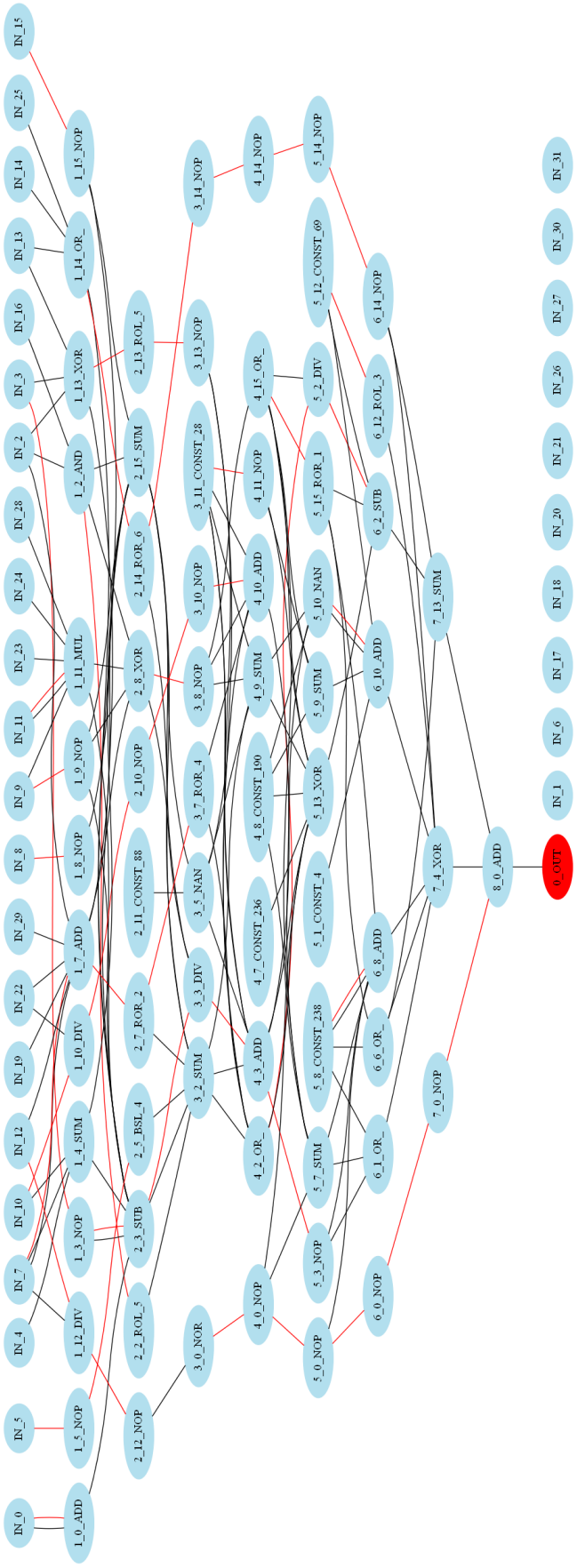
Obrázok A.3: MD5, prediktor č. 1



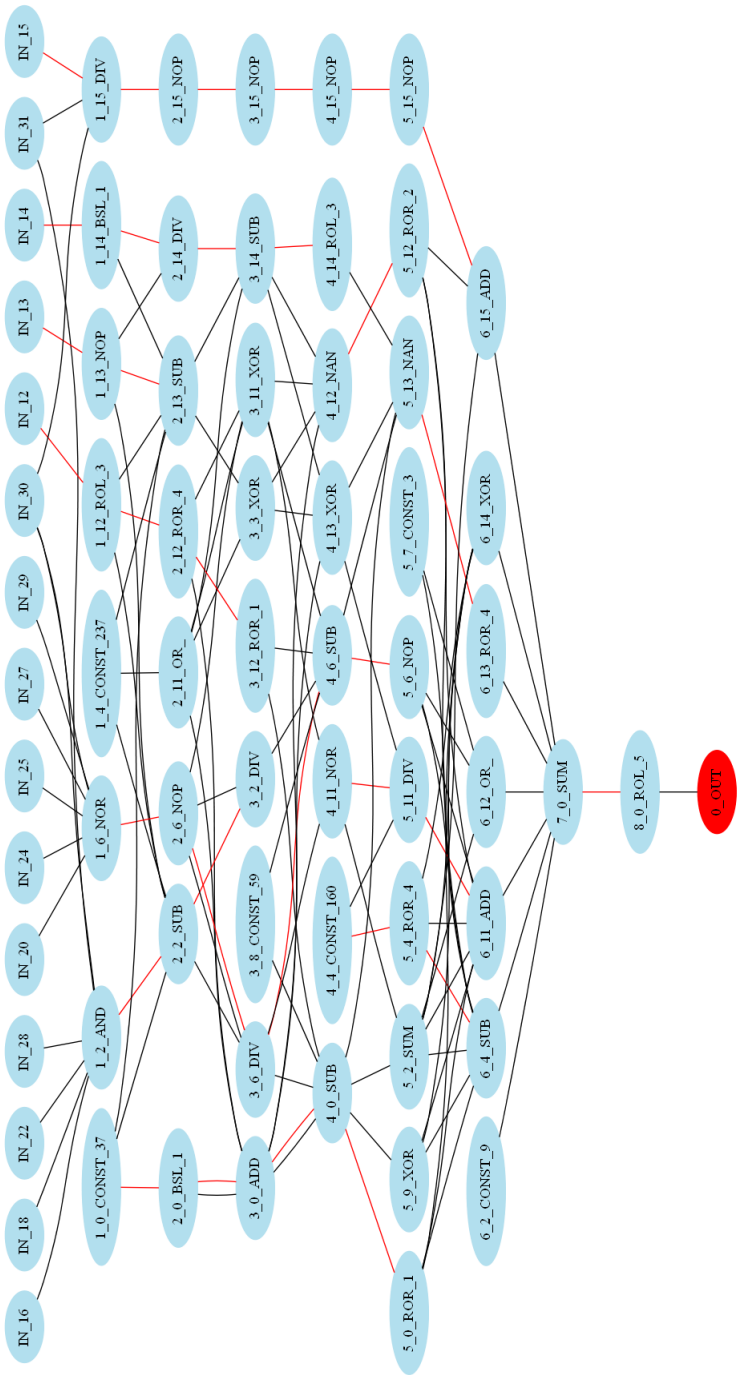
Obrázok A.4: MD5, prediktor č. 2



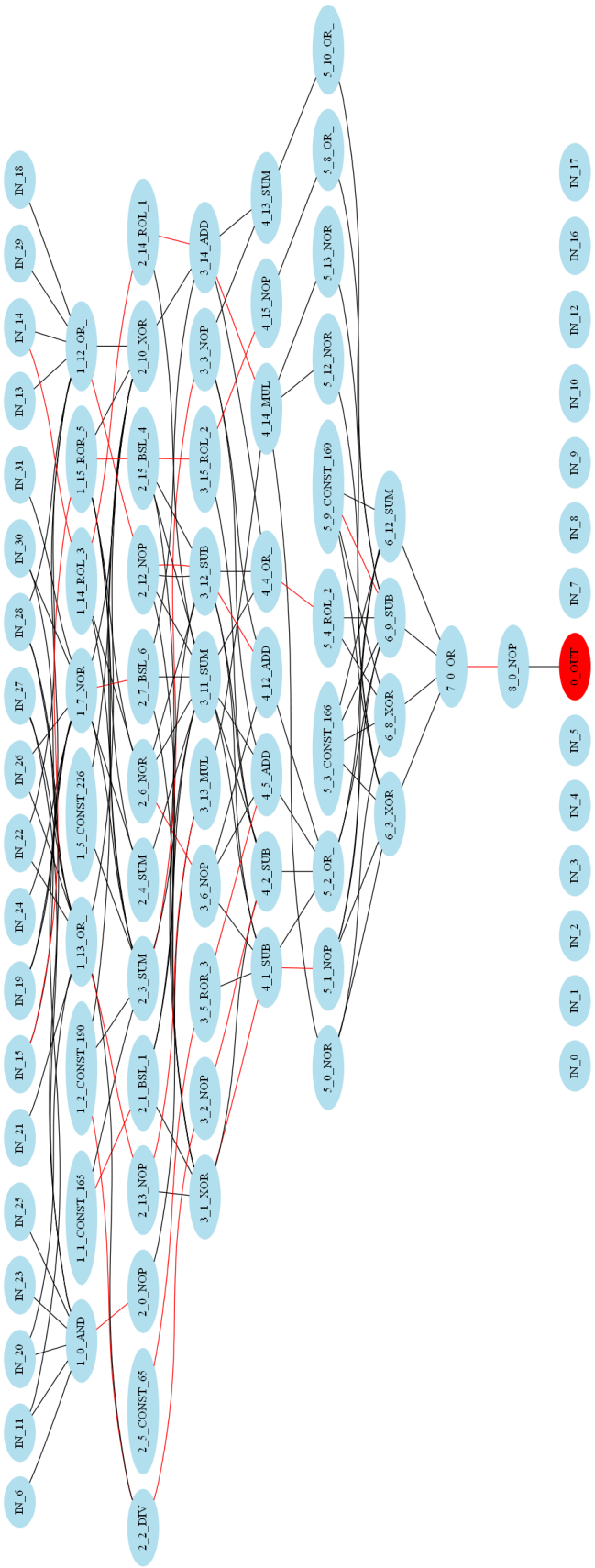
Obrázok A.5: Dynamic SHA, 5 rund, obvod pre hodnotu fitness 0,53



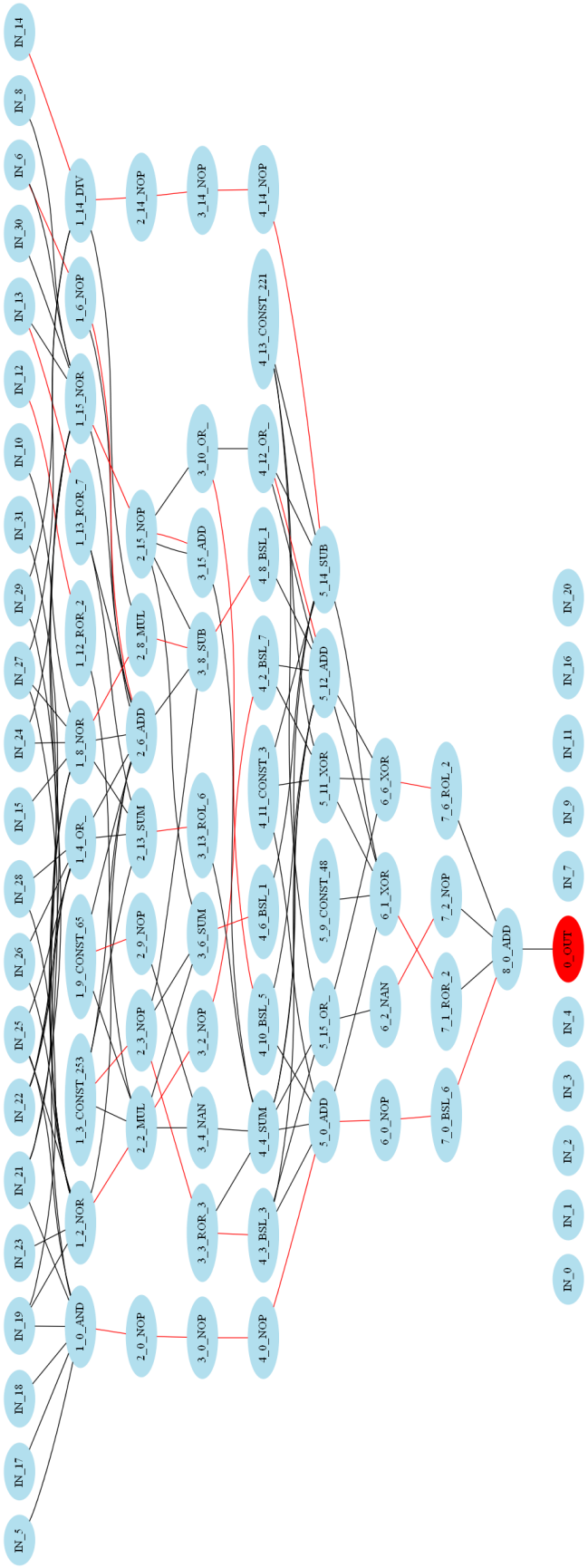
Obrázok A.6: Dynamic SHA, 5 rund, obvode pre hodnotu fitness 0,68



Obrázok A.7: Dynamic SHA, 5 rund, obvod na konci evolúcie



Obrázok A.8: Dynamic SHA, 6 rund, obvod na konci evolúcie



Obrázok A.9: Dynamic SHA, 7 rund, obvod na konci evolúcie

Literatúra

[1] Kerckhoff, A.: *Kerckhoffov princíp*, <http://artofinfosec.com/335/crypto-kerckhoffs-principle/> . 2.1.1

[2] Pavlovič, J.: *Návod k modulu xslt2*, 2006, <http://www.fi.muni.cz/~xpavlov/xml/> . 1

[3] Preneel, B.: *Analysis and Design of Cryptographic Hash Functions*, , Február 1993, , <http://homes.esat.kuleuven.be/~preneel/phd_preneel_feb1993.pdf> . 2.1, 2.1.1, 2.1.2, 2.1.4, 3.1.2

[4] Wang, X. a Yin, Y. a Yu, H.: *Finding Collisions in the Full SHA-1*, Springer Berlin / Heidelberg, 2005, 978-3-540-28114-6, <http://dx.doi.org/10.1007/11535218_2> . 1

[5] Bellare, M. a Kilian, J. a Rogaway, P.: *The Security of the Cipher Block Chaining Message Authentication Code*, Journal of Computer and System Sciences, Volume 61, Issue 3, December 2000, <http://www.sciencedirect.com/science/article/pii/S002200009991694X> . 2

[6] *GAlib, A C++ Library of Genetic Algorithm Components*, <http://lancet.mit.edu/ga/> . 4.2

[7] *Graphviz - Graph Visualization Software, domovská stránka*, <http://www.graphviz.org/> . 1

[8] Hellman, M.: *A Cryptanalytic Time-Memory Trade-Off*, IEEE transactions on Information Theory, Vol. 26, 1980, <http://caislab.kaist.ac.kr/lecture/2010/spring/cs548/basic/B01.pdf> . 3.3

[9] Klíma, V.: *Finding MD5 Collisions – a Toy For a Notebook*, , 5. Marec 2005, , <http://cryptography.hyperlink.cz/md5/MD5_collisions.pdf> . 3.4

[10] *Príklad útoku pomocou Rainbow tables*, Wikipedia, 2006, <http://en.wikipedia.org/wiki/File:Rainbow_table2.svg> . 3.3

[11] *Quantum Random Generator Service*, , <https://qrng.physik.hu-berlin.de/> . 4.3

[12] *Sensor Security Simulator (S3), domovská stránka*, <http://www.fi.muni.cz/~xsvenda/s3.html> . 4.2

[13] *ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions*, <http://csrc.nist.gov/groups/ST/hash/documents/SHA3-C-API.pdf> . 4.4

[14] Naor, M. a Yung, M.: *Universal One-Way Hash Functions and their Cryptographic Applications*, Proceedings of the twenty-first annual ACM symposium on Theory of computing, Seattle, Washington, United States , 1989, 0-89791-307-8, <http://doi.acm.org/10.1145/73007.73011>. 2.1.3

[15] Wang, X. a Feng, D. a Lai, X. a Yu, H.: *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, 17. August 2004, <http://eprint.iacr.org/2004/199.pdf>. 3.4