KAPITOLA 1

Štatistické testovanie náhodnosti

Náhodné čísla zohrávajú dôležitú úlohu v rôznych odvetviach informatiky. Veľkú rolu majú napríklad v kryptografii, pretože cieľom šifrovania dát je aj nemožnosť zistiť, či sú dáta výstupom zo šifrovacej funkcie, alebo len náhodná sekvencia. Aby sme vedeli povedať, či šifrovacia funkcia spĺňa toto kritérium, potrebujeme nástroj, ktorým keď preskúmame dáta zistíme, či sú náhodné alebo nie.

Najrozšírenejší nástroj na testovanie náhodnosti sú tzv. štatistické sady. Každá sada obsahuje testy, ktoré sú potom zoskupené do batérií zostavených z niekoľkých testov. Každý štatistický test skúma požadovanú vlastnosť na vstupných dátach. Z celej batérie sa po získaní výsledku z každého testu vyhodnotí, aká je pravdepodobnosť, či sú vstupné dáta náhodné.

1.1 NIST STS

Najznámejšia zo štatistických sád na štatistické testovanie náhodnosti *Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* [Ruk+00], ktorá vznikla v národnom inštitúte štandardov a technológií (NIST), používa vo svojej testovacej sade množinu 15-tich testov, ktoré boli zostavené na základe predstavy o tom, ako by malo náhodné číslo vyzerať. Napríklad pri zápise v binárnej sústave by mal byť počet cifier 1 a 0 približne rovnaký. Niektoré z testov majú dokonca viac variant a parametrov, takže v konečnom dôsledku sa nad dátami spustí niekoľko násobne viac testov. Avšak interpretácia výsledkov nie je triviálna. Každý test sa spustí nad viacerými sekvenciami. Ako výsledok testu, pre každú sekvenciu je hodnota p-value, z intervalu $[0, 1]$, ktorá značí pravdepodobnosť, že by takúto sekvenciu vygeneroval aj naozaj náhodný generátor. Napríklad pre 1000 sekvencii dostaneme ku každému testu 1000 p-values. Na určenie výsledku sa používajú 2 metódy:

- **Uniformné rozloženie p-values po celom intervale** $[0, 1]$
  Skúma sa rovnomerné rozloženie po celom intervale. Interval sa rozdelí na 10 častí a teda v každej časti by malo v našom prípade skončiť približne 100 hodnôt.
- **Pomer prejdených testov**
  Na určenie výsledku potrebujeme hodnotu hladiny významnosti (α) a interval do ktorého musí pomer spadnúť (interval je vypočítaný na základe hodnoty α). Každá z 1000 p-value získaná z jedného testu je porovnaná s hodnotou α. Ak je hodnota menšia, test pre jednu sekvenciu neprešiel, ak je väčšia, test prešiel. Pomer sekvencií ktoré testami prešli, ku všetkým, by mal ležať vo vypočítanom intervale.

Podľa článku *On the Interpretation of Results from the NIST Statistical Test Suite* [Sýs+15] je vysoká pravdepodobnosť, až 80%, že aj výstup z naozaj náhodného generátora neprejde niekoľkými testami. Autori spomenutého článku ukázali, že dáta sa dajú považovať za náhodné aj vtedy, ak neprejde (to znamená p-value je menšia ako $\alpha = 1\%$) 6 testov.

1.2 Dieharder

Dieharder [Bro04] je sada vytvorená Robertom G. Brownom na Duke univerzite za účelom zrýchliť a zjednodušiť spúšťanie testov tak, aby ich mohol rýchlo a jednoducho spustiť každý, kto potrebuje o svojich dátach zistiť, či sú náhodné. Testovacia sada je následníkom Diehard-u [Mar95], avšak testy sú upravené a začlenené do rovnakej štruktúry. Taktiež sú do nej pridané testy z iných sád alebo od iných samostatných autorov. Vo verzii 3.31.1 z roku 2003 sa nachádza 31 testov, z toho 17 pochádza z pôvodnej sady diehard, 3 zo sady STS NIST (autori očakávajú, že raz bude obsahovať všetky testy) a zvyšných 11 z rôznych zdrojov, napríklad od samotného autora R. G. Browna.

1.3 TestU01

Tvorcom Test-U01 [LS07] je Pierre L'Ecuyer, ktorý pôsobí na univerzite v Montreale. Táto sada obsahuje množstvo testov, ktoré sa dajú spúšťať rôznymi spôsobmi, či už formou batérií, alebo samostatne. Testy sú implementované v jazyku ANSI C. Knižnica je rozdelená do viacerých modulov, popis modulov je k dispozícii v dokumentácii [LEc09], jedným z nich sú aj testovacie batérie. Sada obsahuje viac batérii testov, každá batéria má svoje určenie.

- **Small crush**
  Vytvorená tak, aby čo najrýchlejšie poskytla výsledok, preto neobsahuje veľa testov. Slúži na testovanie generátorov náhodných čísel.
- **Crush**
  Narozdiel od Small crush obsahuje viac testov. Zaberie teda viac času, avšak ak všetky testy prejdú, môžme si byť istejší pravdivosťou výsledku. Takisto ako small crash slúži na testovanie generátorov.
- **Big crush**
  Ešte väčšia a pomalšia batéria ako crush a small crash.
- **Alphabit**
  Primárne určená na hardware generátory, na vstupe môže brať aj jeden binárny súbor.
- **Rabbit**
  Takisto ako Alphabit môže mať ako vstup binárny súbor.
- **A ďalšie**
  Obsahuje ešte iné batérie, ktoré simulujú batérie spomenuté vyššie, napríklad

PseudoDIEHARD, ktorá simuluje batériu DIEHARD [Mar95]. Alebo batéria FIPS_140_2, ktorá napodobňuje STS od NIST.

KAPITOLA 2

Framework EACirc

Testovanie náhodnosti štatistickými testami (kapitola 1) má však aj svoje nevýhody. Pre zjednodušenie si predstavme, že sa v batérii nachádza iba jeden test, ktorý testuje, či je počet núl a jednotiek približne rovnaký. Potom nie je zložité vytvoriť sekvenciu, ktorá testom prejde, napríklad postupnosť, v ktorej sa pravidelne striedajú nuly a jednotky, avšak je veľmi malá pravdepodobnosť, že by takáto sekvencia bola výstupom z naozaj náhodného generátora. Nevýhodou štatistických sád je, že testy, ktoré obsahujú, dokážu odhaliť iba nezrovnalosti, na ktoré boli naprogramované. Preto sa v štatistických sadách nachádza veľké množstvo testov, avšak každý test pridáva iba jednu vlastnosť, ktorú kontroluje. Avšak náhodnosť, neznamená spĺňať presne danú množinu vlastností. Z toho vyplýva, že výsledok zo štatistických testov nemusí vždy garantovať, že sú dáta naozaj náhodné respektíve nenáhodné. Tento nedostatok rieši alternatívny prístup, *Framework EACirc* [Ukr13].

Prístup EACircu je oproti štatistickým testom úplne odlišný. Nesnaží sa priamo určiť či sú skúmané dáta náhodné, namiesto toho hľadá funkciu, ktorá určí či na vstupe dostala naozaj náhodné dáta, alebo skúmané dáta. Na základe nájdenia respektíve nenájdenia takejto funkcie vyhlasuje, či sú skúmané dáta náhodné. Z toho vyplýva, že použitie kvalitných náhodných dát, je veľmi dôležitou súčasťou EACircu.

2.1 Princíp fungovania

Vytváranie hore zmienenej funkcie pripomína pomyselnú skladačku. Zatiaľ čo štatistické testy sa dajú prirovnať k hotovej skladačke, s ktorou sa nedá hýbať. EACirc obsahuje iba samotné komponenty, z ktorých sa dá výsledná skladačka poskladať akýmkoľvek spôsobom. Najdôležitejšou úlohou EACircu je zložiť tieto komponenty do jedného celku tak, aby výsledná skladačka splňovala požadované vlastnosti, teda aby funkcia opísaná touto skladačkou vedela rozlišovať medzi náhodnými dátami a skúmanými dátami. Na poskladanie používa samovzdelávací, genetický algoritmus (detailný pohľad v sekcii 2.2), ktorý najprv náhodne poskladá ľubovoľné kocky na seba a potom sa skladačku snaží malými zmenami vylepšovať.

Cieľom EACircu je využiť tento prístup na to, aby ním z jednoduchých funkcií (vysvetlenie funkcií v sekcii 2.3) vytvoril postupnosť, ktorá dokáže rozlíšiť či na vstupe dostala náhodné dáta. S týmto prístupom sa spája viac výhod, napríklad:

- **Na vytváranie testov nie je potrebná žiadna ľudská aktivita**
  Testy zo štatistických sád bývajú založené na matematických problémoch, nad

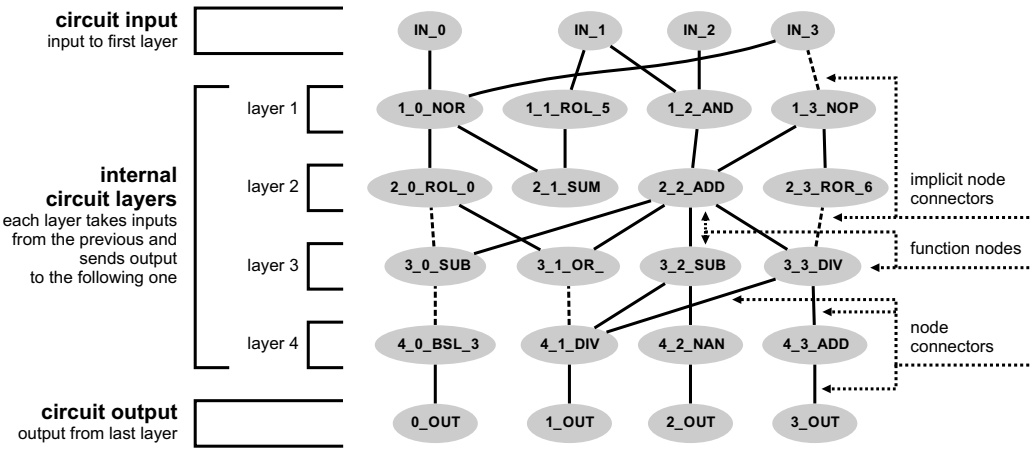
ktorými museli ľudia stráviť množstvo času.

- **Testovanie aj zatiaľ nepoznaných problémov**
  Neexistujú testy na všetky vlastnosti nenáhodných dát, ale EACirc dokáže testovať teoreticky čokoľvek k čomu genetika dospeje.

Takisto ako niektoré štatistické batérie (kapitola 1.), aj EACirc obsahuje integrované generátory prúdov dát, napríklad niekoľko hašovacích funkcií z rodiny SHA3 [Dub12], alebo prúdové šifry eStream [Pri12]. Tiež obsahuje kandidátov zo súťaže CAESAR, ktorých pridal vo svojej diplomovej práci [Ukr16] Martin Ukrop. EACirc potrebuje na svoj beh XML súbor v ktorom sa nachádza konfigurácia, napríklad nastavenie prúdov, počet rúnd atď. Preto Ľubomír Obrátil vytvoril nástroj OneClick [Obr15]. Ako už predznamenáva názov, jedná sa o nástroj, ktorý zjednodušuje prácu s EACircom, napríklad generuje konfiguračné súbory alebo vyhodnocuje výsledky. EACirc podporuje aj zrýchlenie výpočtu pomocou nVidia CUDA technológie, toto rozšírenie doplnil vo svojej bakalárskej práci [Nov15] Jiří Novotný.

2.2 Genetika

Algoritmus použitý v EACircu je založený na biologicky inšpirovanom samovzdelávacom algoritme. V tejto kapitole si objasníme princíp načrtnutý v sekcii 2.1. Kocky skladačky sú v skutočnosti uzly grafu, spojiť dve kocky znamená vytvoriť medzi nimi v grafe cestu. Graf je rozdelený do horizontálnych vrstiev, ktoré sú poskladané na seba, kde v každej z nich sa nachádza niekoľko uzlov. Cesty vedú len smerom zhora nadol, a to len medzi vrstvami idúcimi bezprostredne za sebou (obrázok 2.1). Prvá vrstva je vstupná a posledná výstupná. Keďže sa jedná o biologický algoritmus tento graf sa tiež označuje pojmom jedinec alebo obvod.



Obrázok. 2.1: Vizualizáciu obvodu z bakalárskej práce [Ukr13] Martina Ukropa.

Celý priebeh algoritmu spočíva v nasledujúcich krokoch:

1. **Náhodné vygenerovanie obvodov**
   Vytvorí sa tzv. populácia náhodných jedincov. Veľa z nich bude neúspešných, avšak nájdu sa aj takí, ktorí budú od ostatných lepší.

2. **Určenie úspešnosti**
   Úspešnosť sa vypočíta pomocou tzv. funkcie vhodnosti, ktorá je pre správny priebeh veľmi dôležitá.

3. **Vyradenie neúspešných obvodov**

4. **Sexuálne skríženie najúspešnejších jedincov**
   Skrížením dostaneme novú populáciu jedincov. Cieľom kríženia je dosiahnuť novú silnejšiu generáciu, založenú len na tých najlepších jedincoch z predchádzajúcej populácie.

5. **Náhodná mutácia niektorých jedincov**
   Aby sa zabránilo zaseknutiu sa v lokálnom maxime, je potrebné urobiť nejakú náhodnú mutáciu, napríklad odobrať cestu, alebo zmeniť niektorý z uzlov. Mutácia prebieha tak, že sa postupne prechádza obvodom a pri každom uzle respektíve hrane sa z nejakou pravdepodobnosťou vykoná mutácia.

6. Kroky 2-5 su prevádzané v cykle až kým sa nedosiahne požadovaná úspešnosť alebo kým sa nevyčerpá určený počet generácií.

Podstata genetiky je nájsť obvod, ktorý vie rozlišovať medzi naozaj náhodnými dátami a skúmanými dátami. Ak sa takýto obvod nájde, znamená to, že EACirc objavil v skúmaných dátach niečo, čo sa v naozaj náhodných dátach vyskytuje zriedkavo. To znamená, že dáta zrejme nebudú náhodné.

2.3 Typy uzlov

V tejto sekcii si vysvetlíme aká je vlastne podstata samotného fungovania obvodov. Ako už bolo spomenuté v sekcii 2.2 každý jedinec sa skladá z horizontálnych vrstiev, kde v každej vrstve sa nachádzajú uzly. Každý uzol nesie v sebe informáciu, uloženú na 4 Bytoch, kde na prvom byte je uložené číslo funkcie ktorá sa má vykonať. Na ostatných miestach sú potom uložené voliteľné parametre. Avšak nie všetky funkcie tieto parametre využívajú. Jedná sa o nasledujúce funkcie:

- **Bitové operátory**
  AND, OR, XOR, NOR, NAND, ROTL, ROTR, BITSELECTOR
- **Aritmetické funkcie**
  SUM, SUBS, ADD, MULT, DIV
- **Operátor identity**
  NOP
- **Operátor na priame čítanie zo vstupu**
  READX

Prvá vrstva je vstupná, to znamená, že sa do nej nalejú dáta. Následne dáta prebublávajú smerom nadol, každý uzol dostane na vstupe výstup z niekoľkých uzlov z predchádzajúcej

vrstvy, vykoná funkciu, ktorej referenciu má v sebe uloženú a výstup pošle ďalšiemu respektíve ďalším uzlom v nasledujúcej vrstve.

Použitím jednoduchých funkcií sa dá v konečnom dôsledku docieliť podobnému testu ako sa vyskytuje v štatistických testoch (kapitola 1.). Avšak použitie genetického algoritmu prináša aj možnosť vymyslieť lepšie a silnejšie testy ako sa nachádzajú v batériách. Preto je našim úmyslom vytvoriť pre genetiku čo najlepšiu situáciu na skonštruovanie výsledného obvodu. Následkom je myšlienka, nepoužívať v uzloch iba jednoduché funkcie (napríklad AND, OR atď.), ale vykonať v uzle niečo zložitejšie, napríklad inštrukcie vybraté z programu, ktorý vygeneroval testovaný prúd dát (viac v kapitole 3.).

2.4 Interpretácia výsledkov

EACirc má pre každý beh hypotézu, či sú skúmané dáta náhodné alebo nie. Na konci behu sa táto hypotéza buď potvrdzuje alebo zamieta. Ak sa hypotéza potvrdí, znamená to, že dáta sú náhodné, ak nie, tak EACirc našiel obvod, ktorý vie rozlíšiť medzi skúmanými a náhodnými dátami. To či sa hypotéza zamietne alebo potvrdí sa rozhoduje podľa p-value, čo je hodnota, ktorá je výsledkom každého behu, a hladiny významnosti, čo je hodnota špecifikovaná v konfiguračnom súbore a určuje ktorá je najväčšia hodnota, pre ktorú sa hypotéza ešte zamieta. To znamená, že po získaní p-value sa táto hodnota porovná s hladinou významnosti, ak je p-value menšia, hypotéza sa zamieta, inak sa potvrdzuje.

Keďže celý algoritmus je náhodný a nie vždy sa musí genetike podariť zostrojiť úspešný obvod, existuje istá pravdepodobnosť, takisto ako pri štatistických testoch, že EACirc aj náhodné dáta označí za nenáhodné a naopak. Preto sa výpočty vždy počítajú na viac behov, vždy s náhodným semienkom. Výsledok takýchto výpočtov bude tzv. proportion, čo znamená pomer pre nás úspešných behov ku všetkým behom, teda percentuálne zastúpenie takýchto behov. Pre nás úspešné behy sú také, v ktorých bol EACirc úspešný v hľadaní obvodu, to znamená v ktorých bola hypotéza zamietnutá. Ak je proportion menšia ako hladina významnosti, väčšinou 5%, znamená to, že EACirc nebol úspešný v hľadaní obvodu a teda skúmané dáta sú náhodné. Čím vyššie číslo proportion tým vo viac behoch bol EACirc úspešný v hľadaní obvodu.

KAPITOLA 3

Simulátor Java bytecodu

V tejto kapitole si predstavíme nový typ uzlov, ktoré nevykonávajú iba jednoduchú funkcionalitu (napr. *AND, OR, XOR*), ale emulujú časť z dissasemblovaného Java bytecodu. Dôvod bol spomenutý už v predchádzajúcej kapitole v sekcii 2.3. Zjednodušene potrebujeme pomôcť genetike, aby mohla čo najjednoduchšie vytvoriť výsledný obvod, ktorý bude mať čo najlepšiu úspešnosť. Základná myšlienka je taká, že obvod, ktorý v uzloch vykonáva inštrukcie, ktoré boli vybrané z implementácie kryptografickej funkcie, by mal mať vyššiu šancu rozlíšiť medzi náhodnými dátami a výstupom z tejto funkcie.[1] Jedným z hlavných cieľov tejto bakalárskej práce je overiť, či je táto myšlienka pravdivá.

3.1 Motivácia za použitím Java bytecodu

Java je na jednu stranu vysoko úrovňový jazyk, čo znamená, že je jednoduchší na učenie a tým pádom aj rozšírenejší, no na druhú stranu je jednoduché z neho dostať nízko úrovňový binárny kód, ktorý sa dá interpretovať. Vďaka jej rozšírenosti by mala byť samozrejmosťou dostupnosť implementácie množstva pseudo náhodných generátorov a šifrovacích funkcií. Z tohoto pohľadu sa Java javí ako veľmi dobrý jazyk pre použitie popísané v tejto kapitole.

3.2 Vysvetlenie skratky JVM (Java Virtual Machine)

JVM [MD97] je software, čo spája kód naprogramovaný v Jave a hardware konkrétneho počítača, na ktorom tento kód spúšťame. Vďaka nemu je Java multiplatformová, pretože Javu je možné spustiť všade tam, kde beží JVM. Samotný JVM má podobné princípy ako bežný procesor, teda napríklad obsahuje zásobník, registre a vie vykonávať konečnú množinu inštrukcií.

Fungovanie JVM nie je nič zložitého, jednoducho obsahuje implementáciu všetkých inštrukcií a po načítaní bytecodu vykonáva jednu inštrukciu po druhej. Avšak poznať detailné fungovanie JVM nie je pre účely našej práce dôležité. Dôležité je vedieť, že aj nás JVM simulátor vykonáva inštrukcie z bytecodu. Avšak s tým rozdielom, že vykonáva náhodný kus inštrukcii, vykonáva ich v rámci uzlov EACircu a nepozná úplne všetky

1. V porovnaní s EACircom, ktorý používa bežné uzly.

inštrukcie, ktoré obsahuje klasický JVM. Implementované sú zatiaľ iba tie inštrukcie s ktorými sme sa stretli v niektorom s použitých bytecodov a boli v rámci nášho simulátora implementovateľné. Ďalší rozdiel je napríklad v používaní zásobníka, v klasickom JVM má každá funkcia svoj vlastný lokálny zásobník, ktorý sa používa na predávanie argumentov, lokálne premenné atď. zatiaľ čo náš simulátor používa zásobník ako globálne úložisko, na ktoré sa ukladajú hodnoty počas celého jedného výpočtu, prislúchajúceho jednému uzlu.

3.2.1 Bytecode

Bytecode je označenie pre súbor, ktorý je výstupom po skompilovaní Java kódu. Má presne danú štruktúru, ktorú vie vykonávať každý JVM. Je to v podstate množina funkcií, kde každá funkcia obsahuje niekoľko inštrukcií. Každý riadok z inštrukciou obsahuje číslo inštrukcie, názov inštrukcie, prípadne parametre pre inštrukciu, napríklad inštrukcia *bipush* berie ako parameter číslo, ktoré vloží na zásobník.

3.3 Princíp fungovania emulácie

Fungovanie JVM simulátora je proces, ktorý sa skladá z viacerých bodov, avšak nie je to súvislý proces, jedná sa iba o obsluhu JVM uzlov. Zvyšok EACircu funguje tak ako pri bežných uzloch. V tejto kapitole si prejdeme celý proces krok po kroku, od zvolenia uzlu za JVM uzol až po vykonávanie konkrétnych inštrukcií a výpočet výsledku.

3.3.1 Načítavanie bytecodu

Na začiatku behu EACircu sa musí JVM simulátor nainicializovať, čoho súčasťou je načítavanie bytecodu zo súboru. Funkcie a inštrukcie z tohoto súboru sa budú používať počas celého jedného behu. Jeho názov je uložený v konfiguračnom súbore v elemente *JVM_FILENAME*. Každá funkcia je následne uložená v spojovanom zozname a má priradené unikátne číslo a jej prislúchajúce inštrukcie. Takýmto spôsobom sa postupne načíta celý bytecode. Ak sa pri načítavaní vyskytne nejaká chyba, napríklad neznáma inštrukcia alebo zlá štruktúra bytecodu, vypíše sa chyba a vykonávanie programu skončí.

V bytecode existuje množstvo inštrukcií. Niektoré z inštrukcií vyžadujú na svoje vykonanie aj argumenty, preto sa v štruktúre nachádza aj možnosť uloženia až dvoch argumentov. Napríklad inštrukcia *BIPUSH*, má ako argument číslo, ktoré pri vykonávaní vloží na zásobník.

3.3.2 JVM uzly a voľba parametrov

Každý uzol v EACircu obsahuje 4 Bytovú informáciu (sekcia 2.3). JVM uzol využíva celé 4 Byty a to nasledovne:

- 1. Byte: tu je uložené, že sa jedná o JVM uzol, v súčastnosti číslo 19,
- 2. Byte: číslo funkcie, ktorej inštrukcie sa budú vykonávať,
- 3. Byte: číslo riadka, na ktorom sa nachádza inštrukcia, od ktorej sa začína výpočet,
- 4. Byte: počet inštrukcií, koľko sa má vykonať. To znamená, že sa vykonávajú inštrukcie od tej na riadku z parametru číslo 3 po inštrukciu na riadku, ktorý vznikne spočítaním 3. a 4. parametra. Toto obmedzenie však neplatí pri vykonávaní funkcie zavolanej špeciálnymi inštrukciami *INVOKE*[2] (viac v podsekcii 3.3.3).

Pri tvorbe obvodu sa pri každom uzle EACirc náhodne rozhoduje, akú funkciu bude plniť. Do EACircu bola doplnená funkcia, ktorá sa volá v prípade, že voľba vyberie, že sa jedná o JVM uzol. Táto funkcia do uzla dopĺňa parametre, popísané vyššie. Parametre sa nemôžu voliť náhodne z toho dôvodu, že každý JVM uzol, ktorý sa nakoniec bude nachádzať v obvode, musí byť validný. To znamená funkcia s číslom v parametri 2 musí existovať aj so začiatočnou inštrukciou a dostatkom inštrukcií na vykonávanie podľa posledných dvoch parametrov.

3.3.3 Vykonávanie inštrukcií

Tak ako reálny Java virtual machine aj JVM simulátor obsahuje zásobník. Na začiatku sa naň vložia všetky vstupy, ktoré vykonávaný uzol má. Okrem zásobníka obsahuje aj štruktúru, ktorá určuje stav procesora, teda ktorá funkcia sa vykonáva a na ktorom riadku. Pri požiadavke na vykonanie JVM uzlu sa najprv vyplní táto štruktúra tak, že obsahuje pointer na funkciu a číslo prvej inštrukcie, ktoré sa vybralo z 3 parametru uzlu. Následne sa v slučke emulujú všetky ostatné inštrukcie, s tým, že po každom úspešnom behu sa číslo inštrukcie zdvihne o jedna.

Existujú však aj špeciálne inštrukcie, po vykonaní ktorých sa nepokračuje bežnou cestou, teda pokračovaním na ďalšiu inštrukciu. Napríklad inštrukcie, ktoré preskakujú na iné inštrukcie v rámci funkcie. Jednou z nich je *IF_ICMPGE*, avšak existuje veľa podobných inštrukcií[3] na vetvenie programu, kde sa po splnení podmienky skáče na konkrétnu inštrukciu, ktorej číslo sa nachádza v argumente inštrukcie. Alebo inštrukcia *GOTO*, ktorá automaticky preskočí na požadované miesto. Ďalšie špeciálne inštrukcie sú tie začínajúce na *INVOKE*[4], po ktorých sa síce pokračuje na ďalšiu inštrukciu, avšak až po vykonaní všetkých inštrukcií inej funkcie, ktorá je špecifikovaná v argumentoch inštrukcie.

3.3.4 Výsledok uzlu

Po vykonaní všetkých inštrukcií je výsledkom uzlu *XOR* hodnôt na zásobníku.

2. Medzi inštrukcie *INVOKE* patria: *INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL.*
3. Sú to napríklad: *IFEQ, IFNE, IFGE, IFLE, IF_ICMPGE, IF_ICMPLE, IF_ICMPNE, IF_ICMPEQ, IF_ICMPLT, IF_ICMPGT, IF_ACMPEQ.*
4. Medzi inštrukcie *INVOKE* patria: *INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL.*

3.4 Problémy spojené s implementáciou

Jeden z problémov je nemožnosť uložiť do uzlu viac ako 4B informácie. Keďže prvý Byte je rezervovaný na funkciu, pre naše účely ostávajú iba 3 Byty. Po rozdelení máme teda pre každý z troch parametrov, spomenutých v podsekcii 3.3.2, rozsah [0-255], avšak funkcie v bytecode majú často niekoľko násobne viac inštrukcií. S týmto rozdelením sme schopní vykonať maximálne 255 inštrukcií, a zároveň začať maximálne na riadku 255. Existuje však aj výnimočná situácia, ktorá je prebraná v ďalšom odstavci, kedy je možné vykonať viac ako 255 inštrukcií. V bežnom prípade je teda posledná inštrukcia, ktorú dokážeme vykonať na riadku 510, čo znamená, že akákoľvek inštrukcia za ňou nemôže byť nikdy vykonaná. V súčasnej dobe sa však prerába celá genetika v rámci EACircu a v novej implementácii by mal tento problém zaniknúť, pretože by malo byť v uzle viac miesta pre parametre.

S predchádzajúcim problémom súvisí aj časová náročnosť pri veľkom množstve vykonávaných inštrukcií. EACirc počas jedného behu spracuje veľké množstvo uzlov, preto časová náročnosť rastie pri väčšom množstve vykonávaných inštrukcií veľmi rýchlo. Keďže sa môže vyskytnúť situácia kedy sa môže vykonať aj viac ako 255 inštrukcií, napríklad kvôli inštrukcii *GOTO*, kedy sa stáva, že sa program zacyklí tým, že donekonečna preskakuje niekam nad aktuálnu inštrukciu, tak bolo potrebné limitovať maximálny počet vykonaných inštrukcií na 300. Takisto inštrukcie *INVOKE*[5], môžu zväčšiť počet vykonaných inštrukcií na viac ako 255, a preto je aj v tomto prípade nastavený maximálny počet vykonaných inštrukcií na 300.

3.5 Implementačné rozdiely medzi skutočným JVM a našim JVM

Náš JVM simulátor neobsahuje úplne celú funkcionalitu, ktorá je obsiahnutá v originálnom JVM. Dôvodom je, že nie všetky inštrukcie boli pre nás výhodné na implementáciu, čo sa týka pomeru vynaloženého úsilia a pridanej hodnoty. Preto sa niektoré inštrukcie vždy preskakujú. Napríklad sme úplne vyradili prácu s poľami a objektami. Inštrukcie, ktoré sa preskakujú sú vypísané v prílohe.

3.6 Výhody prístupu

Okrem výhody používať na rozlišovanie dát priamo inštrukcie z ich generátora, je najväčšou výhodou možnosť skonštruovať naozaj komplexný obvod zložený zo zložitejších častí. Otázkou ale je, či je genetika natoľko silná, aby bola schopná zložiť takéto komplexné obvody, pretože s použitím JVM simulátora je naozaj mnohonásobne viac možností ako výsledný obvod poskladať. Preto je ďalšou otázkou aj to, či pre JVM obvody nie je potrebné viac času, a teda viac generácií, na hľadanie výsledného obvodu.

5. Medzi inštrukcie *INVOKE* patria: *INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL.*

3.7 Nevýhody JVM uzlov

Najväčšou nevýhodou je dĺžka výpočtu, avšak je očakávateľná, pretože sa v uzloch vykonávajú časovo zložitejšie výpočty, zatiaľ čo bežný uzol sa dá prirovnať k jednej inštrukcii, JVM simulátor ich vykonáva viac. S vykonávaním inštrukcií je spojená aj réžia, ako napríklad kontrola hodnôt na zásobníku alebo maximálneho počtu inštrukcií. Takže v konečnom dôsledku je jeden beh mnohonásobne dlhší, ako beh z bežnými uzlami.

Niektoré inštrukcie, napríklad *IADD*, ktorá vyberie zo zásobníka dve čísla, spočíta ich a výsledok vloží na zásobník, vyžadujú niekoľko hodnôt na zásobníku, a nie vždy sa tam tieto hodnoty vyskytujú. Z tohoto dôvodu sa stáva, že sa inštrukcie musia preskočiť. To znamená, že ak je zásobník prázdny, žiadna inštrukcia vyžadujúca hodnoty na zásobníku sa nevykoná. Avšak tento nedostatok sa nedá vyriešiť jednoduchou cestou, pretože je spôsobený vykonávaním náhodných inštrukcií, mnohokrát aj zo stredu funkcie. Môže sa teda stať, že vo veľa uzloch sa nevykoná vôbec nič. Potencionálne však máme možnosť vytvoriť zložitý obvod aj keď niekoľko uzlov nerobí nič. Iný pohľad na vec je, že aj obvody, v ktorých je veľa takýchto uzlov, môžu mať v konečnom dôsledku vysokú úspešnosť čo sa týka rozoznávania náhodných dát, preto je len na genetike, aby si vybrala ten správny prístup.

KAPITOLA 4

Experimenty

Hlavným cieľom zavedenia JVM uzlov bolo vylepšiť úspešnosť EACircu. Na otestovanie sme preto museli vyskúšať viacero experimentov, ktoré si predstavíme v tejto kapitole, a porovnať ich výsledky s výsledkami, ktoré dosiahol EACirc s bežnými uzlami, pri rovnakých nastaveniach.

Autorom výsledkov z EACircu s bežnými uzlami, nie som ja, ale Ľubomír Obrátil, ktorému by som sa chcel za výsledky poďakovať.

4.1 Experiment s imitáciou bežných uzlov

Prvý experiment slúžil najme na overenie, či JVM simulátor funguje korektne. Myšlienkou bolo na začiatok nepoužívať bytecode vytvorený z implementácie šifrovacej funkcie, ale skúsiť vytvoriť bytecode, ktorý bude napodobňovať bežné uzly, ktoré sa nachádzajú v EACircu. Očakávané boli podobné výsledky ako dosiahol štandardný EACirc.

4.1.1 Použitý bytecode

Bytecode pre tento experiment sme museli napísať ručne, pretože si to vyžadovalo zamyslenie sa nad tým ako funguje každá jedna funkcia z bežných uzlov. Nie pri všetkých funkciách to bolo jednoduché, pretože prístup JVM simulátora je odlišný od prístupu bežných uzlov.

Najväčší problém bol, že bežné uzly vykonávajú medzi všetkými vstupmi operáciu, napríklad *AND, OR, XOR* atď. V prípade JVM simulátora nie je problém nájsť inštrukcie, ktoré plnia ekvivalentnú funkciu ako bežné uzly. Problém je, že bežné uzly vykonajú operáciu vždy medzi všetkými vstupmi, zatiaľ čo JVM simulátor vykoná náhodné množstvo inštrukcií, preto bolo treba premyslieť, ako ich vykonať dostatok na to, aby sa zvolená inštrukcia vykonala medzi všetkými vstupmi, teda medzi všetkými hodnotami na zásobníku. Tento problém sa dal vyriešiť tým, že by sme vykonávali vždy všetky inštrukcie funkcie, avšak toto riešenie by by bolo v rozpore so základnou myšlienkou JVM simulátora, a tou je vykonávať náhodný kus inštrukcií zo šifrovacej funkcie. Preto sme sa snažili aspoň zvýšiť pravdepodobnosť, že sa vykoná aspoň taký počet inštrukcií aký potrebujeme, a to tak, že sme pre každý typ bežného uzlu vytvorili v bytecode funkciu, a do nej vložili viac krát konkrétnu inštrukciu, a spoliehame sa na genetiku, že si vyberie dostatok inštrukcií na spracovanie všetkých hodnôt na zásobníku.

S týmto súvisí aj to že niektoré funkcie sa nedajú nahradiť jedinou ekvivalentnou inštrukciou, napríklad *ROTL* alebo *ROTR*, ktoré vykonávajú pravú respektíve ľavú rotáciu, ich implementácia obsahuje viac rôznych inštrukcií. V tomto prípade takisto nemôžeme garantovať, že sa vykoná celá funkcia, preto lebo začíname emuláciu od náhodného riadka. Dalo by sa to vyriešiť vykonávaním vždy všetkých inštrukcií vo funkcii, avšak toto riešenie by bolo v rozpore z našou predstavou ako by mal JVM simulátor fungovať, teda vykonávať v uzloch náhodný kus inštrukcií zo šifrovacej funkcie.

Ďalší problém, ktorého výskyt sa ešte znásobil pri použití funkcií, ktoré vykonávajú náhodné množstvo rovnakých inštrukcií, teda potrebujú minimálne toľko hodnôt na zásobníku, aký je ich počet, je čo spraviť ak je na zásobníku nedostatočný počet hodnôt. Inými slovami čo spraviť ak potrebujeme vybrať hodnotu z prázdneho zásobníka. Z tohoto dôvodu sme museli upraviť implementáciu z nasledujúcimi možnosťami.

- **Pri vyberaní hodnoty z prázdneho zásobníka vracať 0**
  Avšak toto riešenie sa ukázalo ako nesprávne, pretože napríklad pre inštrukciu *AND*, platí, že ak vykonáme túto operáciu s 0, výsledok bude vždy 0. To znamená, že v prípade jednej hodnoty na zásobníku, vyberáme dve hodnoty zo zásobníka, z toho druhá hodnota už bude 0 a vykonávame *AND* s 0 a strácame celý doterajší výpočet.
- **V prípade prázdneho zásobníka vracať neutrálnu hodnotu**
  Tento spôsob by vyriešil problém predchádzajúceho riešenia, pretože v prípade, že sa vyberá z prázdneho zásobníka vracala by sa taká hodnota, s použitím ktorej by sa operácia správala ako identita, teda by vrátila vždy hodnotu prvého argumentu. Avšak nevýhodou je, že implementácia takéhoto riešenia, by bola zložitejšia a taktiež je zbytočné vykonávať niečo, čo aj tak nebude mať žiadny dôsledok.
- **Preskakovať inštrukcie, ktoré nemajú dostatok hodnôt na zásobníku**
  Toto riešenie sa ukázalo ako najlepšie aj čo sa týka výkonu, pretože sa nepočítajú zbytočné výpočty, aj čo sa týka vyriešenia pôvodného problému.

4.1.2 Výsledky experimentu

Takto vytvorený bytecode sme použili na rozlíšenie medzi náhodnými dátami a hašovacou funkciou Tangle. Výsledky nepochádzajú z poslednej verzie JVM simulátora, pretože sme ich počítali už pár mesiacov pred napísaním tejto práce. Sivé zafarbenie bunky znamená, že EACirc bol úspešný vo viac prípadoch ako je hladina významnosti, teda 5%, to znamená číslo vo vnútri bunky je väčšie ako 0.05. Je otázne, či takto označovať aj bunky, ktoré majú tesne nad 5%, napríklad 0.55 alebo 0.6, pretože sa môže jednať aj o štatistickú odchýlku, preto sme sa rozhodli, že takto budeme označovať len bunky, ktoré budú mať hodnotu nad 0.07.

Z výsledkov vyplýva, že JVM simulátor síce dokázal rozoznať tie isté rundy ako bežné uzly, avšak s menšou istotou. To znamená, napríklad pre rundu 21, že zatiaľ čo bežné uzly boli schopné v 94% prípadoch nájsť hľadaný obvod, EACirc s JVM uzlami bol to isté schopný spraviť len v 27% prípadoch. Dôvodom menšej úspešnosti sú zrejme nevyriešené problémy spomenuté vyššie.

Výsledky pre hašovaciu funkciu Tangle			
Počet rúnd	**Bežné uzly** (30000 generácií)	**JVM simulátor** (30000 generácií)	**JVM simulátor** (300000 generácií)
21	0.944	0.270	0.461
22	0.932	0.267	0.465
23	0.066	0.036	0.050

Tabuľka 4.1: Výsledky pre hašovaciu funkciu Tangle pri použití rôznych uzlov.

Pri výpočte na 300000 generáciách bolo EACircu poskytnuté 10 krát viac času na hľadanie obvodu, čiže očakávame lepšie výsledky. Avšak ani v tomto prípade, nebol EACirc s JVM uzlami schopný dotiahnuť sa na výsledky, ktoré mali bežné uzly, preto sme sa rozhodli pokračovať v testovaní na ďalší experiment v ktorom už budú použité bytecody zo šifrovacích respektíve hašovacích funkcií.

4.2 Experiment JVM simulátora na malom testbede

Malý testbed je označenie, ktoré používame v našom tíme, na pomenovanie niekoľkých funkcií, patria sem hašovacie funkcie zo sútaže na funkciu SHA-3: *Tangle, Dynamic SHA, Dynamic SHA-2* a prúdová šifrovacia funkcia z kandidátov na funkciu eStream: *Decim*. Experiment spočíval v tom, že sme z týchto všetkých funkcií vytvorili bytecode a spustili všetky kombinácie výpočtov. Teda každú funkciu v kombinácii s každým bytecodom. Motivácia za týmto experimentom je zistiť, či bude EACirc úspešnejší, keď bude v uzloch, pomocou JVM simulátora, vykonávať inštrukcie zo skúmanej šifrovacej funkcie.

4.3 Použité bytecody

Naneštastie sa nám nepodarilo nájsť implementáciu ani jednej z týchto funkcií v Jave, preto sme museli vytvoriť vlastnú. Najjednoduchšie riešenie pre nás bolo prepísať implementáciu z C++, ktorá je dostupná napríklad aj priamo v EACircu, do Javy. K prepisu nám pomohla voľná verzia konvertoru [Tan] z C++ do Javy, od spoločnosti *Tangible Software Solutions Inc.* Po konvertovaní sme museli ešte manuálne upraviť Java súbor tak, aby bol kompilovateľný, pretože niektoré konštrukcie z C++ sa nedali automaticky skonvertovať do Javy, napríklad práca s pamäťou alebo smerníková aritmetika. Takáto implementácia by pravdepodobne nebola funkčná ani korektná, ale na naše účely, teda na využitie inštrukcií, je postačujúca.

Takto vytvorenú implementáciu sme potom jednoducho skompilovali a zo skompilovaného súboru získali konkrétne inštrukcie. Vzniknutý súbor s bytecodom bolo treba

jemne upraviť, aby si s ním poradila funkcia na načítavanie bytecodu, ktorú obsahuje JVM simulátor, napríklad vymazať niektoré tabulátory a podobne. Posledným krokom bolo implementovanie inštrukcií, ktoré JVM simulátor zatiaľ nepoznal, pretože inak by načítavanie zlyhalo.

4.4 Výsledky experimentu

Pre každú funkciu z malého testbedu, sme spočítali 3 rundy, konkrétne poslednú rundu, ktorú zvládne EACirc s bežnými uzlami plus nasledujúca a predchádzajúca. Ďalej sme pre každú rundu sme spočítali kombináciu s každým bytecodom, dohromady teda 4 výpočty na každú rundu, pretože máme 4 bytecody. To znamená, že dohromady sme mali 12 rôznych výpočtov pre každú funkciu. Každý z 12-tich výpočtov sme spustili 1000 krát to znamená dovedna sme mali pre jednu funkciu 12000 behov EACircu. Výpočty sme počítali na metacentre [Tea16], kde sa behy spúšťajú v rámci tzv. úloh. V rámci jednej úlohy sme vypočítali 8 behov za menej ako dva dni, záležalo na akom stroji sa úloha počítala, niektoré zvládli všetkých 8 behov za 10 hodín a niektorým to trvalo aj 30 hodín. To znamená, že sme potrebovali pre jednu funkciu spočítať 1500 úloh, čo sme boli schopný vďaka metacentru urobiť za 2 až 3 dni, pretože nám naraz priradili až 800 strojov.

Pre výsledky platí to isté čo v predchádzajúcom experimente, šedo podfarbené bunky znamenajú, že EACirc bol úspešný a našiel obvod vo viac ako 5% prípadov, plus štatistická odchýlka.

Výsledky pre funkciu Tangle					
Runda	**Bežné uzly** (proportion)	**Tangle** (proportion)	**Dynamic SHA** (proportion)	**Dynamic SHA-2** (proportion)	**Decim** (proportion)
21	0.944	0.605	0.609	0.643	0.453
22	0.932	0.640	0.606	0.666	0.461
23	0.066	0.045	0.058	0.040	0.051

Tabuľka 4.2: Výsledky pre funkciu Tangle, prvý riadok určuje aký bytecode bol použitý.

Výpočet z tabuľky 4.3 je trochu prekvapivý, pretože EACirc je schopný rozlíšiť všetky rundy funkcie *Dynamic SHA*, zatiaľ čo štatistické sady (kapitola 1.) sú schopné rozoznať funkciu len po rundu 7. Štatistické testy boli spustené nad dátami, ktoré vygeneroval generátor, ktorý obsahuje EACirc, z toho vyplýva, že chybu generátora môžeme vylúčiť. To znamená, že buď je funkcia zlá v niečom čo štatistické testy netestujú, alebo sa v implementácii EACircu nachádza chyba, ktorú sme prehliadli.

Výsledky z tabuliek 4.2 až 4.5, sú jemne chaotické, nenašli sme v nich žiadnu spojitosť medzi skúmanou funkciou a bytecodom z tej istej funkcie. Ba dokonca ani jeden z bytecodov sa nedá vyhlásiť za najlepší. Avšak pozitívne je, že narozdiel od experimentu 1 dosiahli JVM uzly porovnateľné výsledky vo väčšine prípadov, ako bežné uzly. Najhoršie výsledky sme dosiahli

Výsledky pre funkciu Dynamic SHA					
Runda	**Bežné uzly** (proportion)	**Tangle** (proportion)	**Dynamic SHA** (proportion)	**Dynamic SHA-2** (proportion)	**Decim** (proportion)
7	1.000	0.996	0.998	1.000	0.899
8	1.000	1.000	1.000	1.000	0.631
9	1.000	1.000	1.000	1.000	0.616

Tabuľka 4.3: Výsledky pre funkciu Dynamic SHA, prvý riadok určuje aký bytecode bol použitý.

Výsledky pre funkciu Dynamic SHA-2					
Runda	**Bežné uzly** (proportion)	**Tangle** (proportion)	**Dynamic SHA** (proportion)	**Dynamic SHA-2** (proportion)	**Decim** (proportion)
10	1.000	1.000	1.000	1.000	1.000
11	0.986	0.967	0.877	0.901	0.753
12	0.059	0.056	0.042	0.057	0.053

Tabuľka 4.4: Výsledky pre funkciu Dynamic SHA-2, prvý riadok určuje aký bytecode bol použitý.

v prípade funkcie Decim, preto sme sa rozhodli, že na tejto funkcii vyskúšame výpočty s viac generáciami. Počet generácií sme zvolili na 300000, to znamená, že každý beh EACircu, bude trvať 10 násobne dlhšie. Z dôvodu veľmi dlhého výpočtu, sme museli znížiť počet behov z 1000 na 400.

Tabuľka 4.6 neobsahuje výsledky pre bežné uzly, pretože bežné uzly sme nepočítali na 300000 generácií, výsledky sa dajú porovnať s výsledkami z tabuľky 4.5, konkrétne stĺpec z označením bežné uzly, avšak treba brať na vedomie, že bežné uzly mali na hľadanie obvodu 10-krát menej času.

Zhrnutie výsledkov, po dopočítaní výpočtov.

4.5 Výpočty so zložitejším bytecodom

Ďalší a zároveň posledný experiment, ktorý sme vyskúšali, bolo použiť v JVM simulátore bytecode, ktorý obsahuje zložitejšie konštrukcie. Otázkou teda je, či bude genetika schopná využiť takéto konštrukcie na zhotovenie lepších obvodov. Na vytvorenie bytecodu sme použili implementáciu šifrovacej funkcie AES [01].

Nastavenia pre tento experiment, boli rovnaké ako v experiemente 2. teda 30000 generácií, 1000 testovacích vektorov a 1000 behov. Takisto sme pre funkciu Decim spočítali aj verziu s 300000 generáciami a 400 behmi.

Výsledky pre funkciu Decim					
Runda	**Bežné uzly** (proportion)	**Tangle** (proportion)	**Dynamic SHA** (proportion)	**Dynamic SHA-2** (proportion)	**Decim** (proportion)
4	0.998	0.137	0.209	0.203	0.114
5	0.802	0.073	0.095	0.079	0.053
6	0.065	0.045	0.046	0.056	0.048

Tabuľka 4.5: Výsledky pre funkciu Decim, prvý riadok určuje aký bytecode bol použitý.

Výsledky pre Decim s použitím 300000 generácií				
Runda	**Tangle** (proportion)	**Dynamic SHA** (proportion)	**Dynamic SHA-2** (proportion)	**Decim** (proportion)
4	TBD	TBD	TBD	TBD
5	TBD	TBD	TBD	TBD
6	TBD	TBD	TBD	TBD

Tabuľka 4.6: Výsledky pre funkciu Decim s použitím 300000 generácií, prvý riadok určuje aký bytecode bol použitý.

Z tabuliek 4.7 až 4.11 vyplýva, že EACirc s bytecodom z funkcie AES sa úspešnosťou vyrovnáva použitím bytecodu z funkcií v predchádzajúcom experimente až na funkciu Decim kde je výrazne lepší, avšak stále sa nevyrovnáva úspešnosti bežných uzlov (tabuľka 4.5). Z toho vyplýva, že môžeme potvrdiť, že použitie zložitejšieho bytecodu je pre genetiku v niektorých prípadoch výhodné.

Tangle	
Runda	**Proportion**
21	0.574
22	0.578
23	0.049

Tabuľka 4.7: Výsledky pre funkciu Tangle s použitím bytecodu z funckie AES.

Dynamic SHA-2	
Runda	**Proportion**
10	1.000
11	0.957
12	0.056

Tabuľka 4.8: Výsledky pre funkciu Dynamic SHA-2 s použitím bytecodu z funckie AES.

Dynamic SHA	
Runda	**Proportion**
7	0.977
8	0.985
9	0.992

Tabuľka 4.9: Výsledky pre funkciu Dynamic SHA s použitím bytecodu z funckie AES.

Decim	
Runda	**Proportion**
4	0.640
5	0.187
6	0.056

Tabuľka 4.10: Výsledky pre funkciu Decim s použitím bytecodu z funckie AES.

Decim - 300000 generácií	
Runda	**Proportion**
4	TBD
5	TBD
6	TBD

Tabuľka 4.11: Výsledky pre funkciu Decim s použitím bytecodu z funckie AES a počtom generácií 300000.

Bibliografia

[01] *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processin Standards Publication 197. 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[Bro04] R. G. Brown. *Dieharder: A Random Number Test Suite*. Ver. 3.31.1. Duke University Physics Department. 2004. URL: http://www.phy.duke.edu/~rgb/General/dieharder.php (cit. 2016-04-03).

[Dub12] O. Dubovec. "Automated search for dependencies in SHA-3 hash function candidates". Bakalárska práca. Fakulta informatiky, Masarykova univerzita, 2012. URL: http://is.muni.cz/th/324866/fi_b_a2/ (cit. 2016-04-20).

[LEc09] P. L'Ecuyer. *TestU01*. Ver. 1.2.3. Université de Montréal. 2009. URL: http://simul.iro.umontreal.ca/testu01/tu01.html (cit. 2016-04-24).

[LS07] P. L'Ecuyer a R. Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33.4 (2007). DOI: 10.1145/1268776.1268777.

[Mar95] G. Marsaglia. *Diehard Battery of Tests of Randomness*. Floridan State University. 1995. URL: http://stat.fsu.edu/pub/diehard/ (cit. 2016-04-20).

[MD97] J. Meyer a T. Downing. *Java virtual machine*. Angličtina. Cambridge, [Mass.] : O'Reilly, 1997. ISBN: 1565921941.

[Nov15] J. Novotný. "GPU-based speedup of EACirc project". Bakalárska práca. Fakulta informatiky, Masarykova univerzita, 2015. URL: http://is.muni.cz/th/409963/fi_b/ (cit. 2016-04-20).

[Obr15] Ľ. Obrátil. "Automated task management for BOINC infrastructure and EACirc project". Bakalárska práca. Fakulta informatiky, Masarykova univerzita, 2015. URL: https://is.muni.cz/th/410282/fi_b/ (cit. 2016-04-20).

[Pri12] M. Prišťák. "Automated search for dependencies in eStream stream ciphers". Diplomová práca. Fakulta informatiky, Masarykova univerzita, 2012. URL: http://is.muni.cz/th/172546/fi_m/ (cit. 2016-04-20).

[Ruk+00] A. Rukhin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. zpr. 2000. URL: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf (cit. 2016-04-02).

[Sýs+15] M. Sýs, Z. Říha, V. Matyáš, K. Márton a A. Suciu. "On the Interpretation of Results from the NIST Statistical Test Suite". In: *Romanian Journal of Information Science and Technology* 18.1 (2015), s. 18–32.

[Tan] Tangible Software Solutions Inc. *C++ to Java Converter*. URL: http://www.tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html (cit. 2016-30-04).

[Tea16] Team Czech NGI. *MetaCentrum. Virtual Organization of the Czech National Grid Organization*. 2016. URL: https://metavo.metacentrum.cz/ (cit. 2016-05-01).

[Ukr13] M. Ukrop. "Usage of evolvable circuit for statistical testing of randomness". Bakalárska práca. Fakulta informatiky, Masarykova univerzita, 2013. URL: http://is.muni.cz/th/374297/fi_b/ (cit. 2016-04-02).

[Ukr16] M. Ukrop. "Randomness analysis in authenticated encryption systems". Diplomová práca. Fakulta informatiky, Masarykova univerzita, 2016. URL: https://is.muni.cz/th/410282/fi_b/ (cit. 2016-04-20).