KAPITOLA 1

Štatistické testovanie náhodnosti

Náhodné dáta zohrávajú dôležitú úlohu v rôznych odvetviach informatiky. Veľkú rolu majú napríklad v kryptografii, pretože jedným z požiadavkov na zašifrované dáta je aj nemožnosť zistiť, či sú výstupom zo šifrovacej funkcie, alebo len náhodná sekvencia. Aby sme vedeli rozoznať, či šifrovacia funkcia spĺňa toto kritérium, potrebujeme nástroj, ktorým keď preskúmame dáta zistíme, či sú náhodné alebo nie. Avšak zistiť, či sú dáta náhodné nie je vôbec jednoduché, pretože vlastnosť naozaj náhodného generátora je, že vygeneruje každú sekvenciu s rovnakou pravdepodobnosťou. To znamená, že každá sekvencia môže byť výstupom z náhodného generátora.

Najrozšírenejším nástrojom na testovanie náhodnosti sú tzv. štatistické sady často nazývané aj štatistické batérie. Každá sada pozostáva z viacerých testov, kde každý test skúma požadovanú vlastnosť na vstupných dátach. Z každého spusteného testu je výstupom výsledok, všetky tieto výsledky sú následne skombinované do jedného konečného výsledku. Keďže sa nikdy nedá určiť na 100 per cent, či sú dáta naozaj náhodné, tak výsledok nemôže byť formou áno respektíve nie. Preto sa výsledok vyjadruje iba ako pravdepodobnosť s akou by naozaj náhodný generátor vygeneroval menej náhodné dáta ako testované dáta [Sýs+15].

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 mala náhodná sekvencia vyzerať. Napríklad pre každú sekvenciu vygenerovanú náhodným generátorom platí, že pri zápise v binárnej sústave je na každej pozícií s rovnakou pravdepodobnosťou 1 alebo 0. Preto je malá pravdepodobnosť, že naozaj náhodná sekvencia bude napríklad obsahovať len samé jedničky, naopak z oveľa vyššou pravdepodobnosťou bude počet 1 a 0 približne rovnaký. Z tohoto dôvodu je jedným z testov, ktorý obsahujú štatistické sady aj test, ktorý testuje či je táto vlastnosť splnená. Niektoré z testov majú dokonca parametre vďaka ktorým sa v konečnom dôsledku nad dátami môže spustiť niekoľko násobne viac inštancií testov. Cez rozhranie na príkazovom riadku sa dá pred spustením nastaviť akákoľvek kombinácia testov, ktorú si želáme spustiť. Avšak interpretácia výsledkov nie je triviálna. Každý test sa spustí nad viacerými sekvenciami. Ako výsledok testu, pre každú sekvenciu je p-hodnota z intervalu $[0, 1]$, ktorá značí pravdepodobnosť, že by naozaj náhodný generátor vygeneroval menej náhodnú sekvenciu.

Napríklad pre 1000 sekvencií dostaneme ku každému testu 1000 p-hodnôt. Na určenie výsledku sa používajú 2 metódy:

- **Uniformné rozloženie p-hodnôt po celom intervale** $[0, 1]$
  Na to aby sa dáta dali považovať za náhodné by mali byť všetky p-hodnoty rovnomerne rozdelené po celom intervale $[0, 1]$. To znamená, že keď si interval rozdelíme na 10 častí podľa hodnoty, teda prvá časť bude obsahovať p-hodnoty z intervalu $[0, 0.1]$, druh časť p-hodnoty z intervalu $[0.1, 0.2]$ atď. v každej časti by malo skončiť rovnaké množstvo p-hodnôt, v našom prípade približne 100.
- **Pomer úspešných testov ku všetkým testom**
  Ďalší spôsob na určovanie výsledku je tzv. proportion, jedná sa o pomer úspešných testov ku všetkým. 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-hodnôt 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-hodnota je menšia ako $\alpha = 1\%$) 6 testov. Autori v článku používali nimi zvolenú množinu 188 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 aj 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, napríklad prostredníctvom 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, kde každá batéria má svoje určenie.

- **Small crush**
  Vytvorená tak, aby čo najrýchlejšie poskytla výsledok, preto neobsahuje veľa testov. Slúži na testovanie generátorov náhodných sekvencií.
- **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. Ale 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. Vo svojej podstate je to nástroj, ktorému nastavíte dva prúdy dát a on sa snaží nájsť medzi nimi rozdiel. Toto vykonáva prostredníctvom vytvárania funkcie, ktorá dokáže určiť, či dostala dáta z prvého respektíve druhého prúdu. Následne je na základe nájdenia respektíve nenájdenia takejto funkcie určená podobnosť týchto dvoch prúdov. Ak chceme aby EACirc fungoval podobne ako štatistické testy, teda aby určoval aká je pravdepodobnosť, že sú preverované dáta náhodné, stačí nastaviť ako jeden z prúdov dát naozaj náhodné dáta. V prípade, že EACirc nenájde rozdiel medzi náhodnými dátami a preverovanými dátami znamená to, že sú tieto dáta pravdepodobne tiež náhodné. Naopak ak sa rozdiel nájde znamená to, že tieto dáta pravdepodobne nie sú náhodné. Z toho vyplýva, že pre korektné fungovanie EACircu, je veľmi dôležité použiť kvalitné náhodné dáta. V opačnom prípade EACirc nebude vedieť ako majú naozaj náhodné dáta vyzerať a preto môže označiť za náhodné aj dáta, ktoré síce nebudú náhodné, ale budú podobné dátam, o ktorých si EACirc myslí, že náhodné sú.

Keďže pri testovaní sa EACirc používa na podobné účely ako štatistické testy, preverované dáta často pochádzajú zo šifrovacej respektíve hašovacej funkcie. Aby bolo jednoduchšie do niektorého z prúdov nastaviť výstup z kryptografickej funkcie, tak EACirc obsahuje niekoľko integrovaných generátorov prúdov. Jedná sa napríklad o kandidátov na funkciu SHA3 [Dub12], alebo o kandidátov na funkciu eStream [Pri12]. Tiež obsahuje kandidátov zo súťaže CAESAR [Ukr16].

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 operácií (vysvetlenie operácií v sekcii 2.3) vytvorila funkciu, ktorá bude vykonávať postupnosť týchto operácií tak, aby dokázala zo vstupu zistiť, či dostala náhodné dáta alebo preverované dáta. S týmto prístupom sa spája niekoľko 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**
  Štatistické sady neobsahujú testy na všetky vlastnosti nenáhodných dát, nad druhej strane EACirc vytvára hľadanú funkciu náhodne, z toho vyplýva, že v nej môže teoreticky testovať čokoľvek k čomu genetika dospeje.

2.2 Genetika

V tejto sekcii si objasníme princíp, ktorý spočíval v skladaní skladačky načrtnutý v sekcii 2.1 a aj to, čo znamená, že je poskladanie tejto skladačky založené na samovzdelávacom, genetickom algoritme.

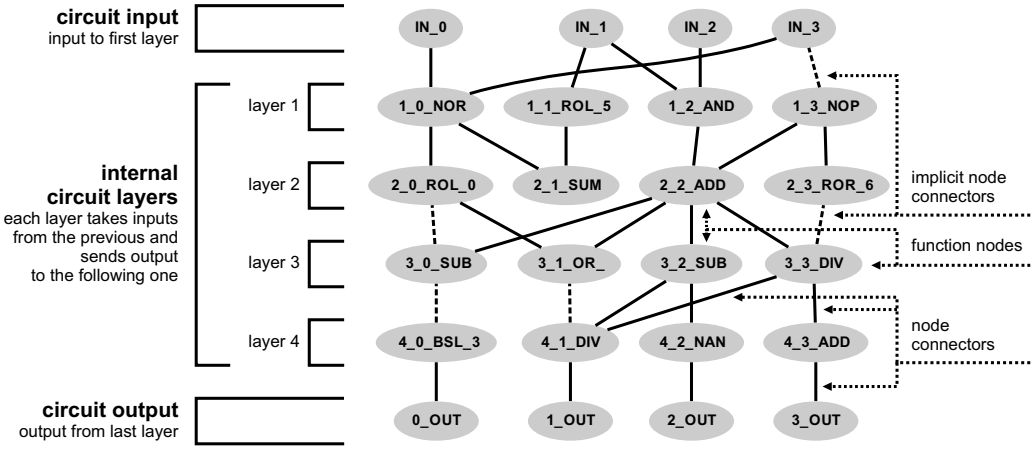
Kocky skladačky sú v skutočnosti uzly grafu. Spojiť dve kocky na seba znamená vytvoriť v grafe cestu medzi uzlami, ktoré sú reprezentované týmito kockami. Graf zoskupený do horizontálnych vrstiev, ktoré sú poskladané na seba, kde v každej z nich sa nachádza niekoľko uzlov. Cesty vedú len smerom zhora nadol, a to len medzi vrstvami idúcimi bezprostredne za sebou (obrázok 2.1). Prvá vrstva je vstupná a posledná výstupná. Keďže sa jedná o genetický algoritmus tento graf budeme označovať aj pojmami jedinec alebo obvod.

TODO: dovysvetlenie genetiky + nejaky zdroj o genetike. 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**



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

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.

Generovanie náhodných hodnôt počas celého algoritmu sa vykonáva pomocou pseudo náhodného generátora. V žiadnom prípade sa však tento generátor nepoužíva na vytváranie prúdu ktorý sa používa na porovnávanie s preverovanými dátami, jedná sa len o hodnoty, ktoré sa používajú pri vytváraní výsledného obvodu. V každom behu sa na začiatku zvolí náhodné počiatočné semienko a na ňom závisia všetky ďalšie vygenerované hodnoty. Použitie pseudo náhodného generátora bolo zvolené kvôli reprodukovateľnosti výpočtov. To znamená, kvôli tomu aby EACirc vracal rovnaké výsledky pre behy, ktoré majú rovnaké nastavenia a počiatočné semienko.

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. 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

Na ostatných miestach v uzle sú potom uložené parametre. Avšak nie všetky funkcie tieto parametre využívajú. Prvá vrstva je vstupná, to znamená, že uzly z tejto vrstvy dostanú na vstupe priamo dáta z jedného z prúdov. Následne dáta prebublávajú smerom nadol, každý uzol dostane na vstupe výstup z niekoľkých uzlov z predchádzajúcej vrstvy. Na vstupoch následne vykoná funkciu, ktorú ma uloženú na prvom Byte v uzle a výstup pošle ďalšiemu respektíve ďalším uzlom v nasledujúcej vrstve.

Použitie jednoduchých funkcií môže v konečnom dôsledku vyústiť do zložitejších konštrukcií. Dokonca sa môže docieliť podobného 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 kódu, 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-hodnoty, 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 dokončení výpočtu sa získaná p-hodnota porovná s hladinou významnosti, ak je p-hodnota 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 bol úspešný v hľadaní obvodu len no veľmi málo prípadoch 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. Tieto uzly vznikli za účelom 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 ako EACirc s bežnými uzlami. 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 inštrukcií, vykonáva ich v rámci uzlov EACircu a nepozná úplne všetky inštrukcie, ktoré obsahuje klasický JVM. Implementované sú zatiaľ iba tie inštrukcie s ktorými sme sa stretli v niektorom z 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 napríklad 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 s inštrukciou obsahuje číslo inštrukcie, názov inštrukcie, prípadne argumenty pre inštrukciu. Napríklad inštrukcia *bipush* berie ako argument 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.

3.3.2 JVM uzly a voľba parametrov

Každý uzol v súčasnej implementácií 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* (viac v podsekcii 3.3.3). Medzi inštrukcie *INVOKE* patria: *INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL*.

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 síce volia náhodne, ale pre zvolené hodnoty musí platiť, že uzol z nich vytvorený, je 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. Sú to inštrukcie začínajúce na *IF* teda napríklad: *IFEQ, IFNE, IFGE* alebo *IFLE*. Tieto inštrukcie slúžia na vetvenie programu, kde sa po splnení podmienky skáče na konkrétnu inštrukciu, ktorej číslo sa nachádza v argumente inštrukcie. Ďalšia inštrukcia, ktorá skáče v rámci funkcie je *GOTO*, avšak narozdiel od inštrukcií *IF* skáče automaticky, bez kontrolovania podmienky. Ďalšie špeciálne inštrukcie sú tie začínajúce na *INVOKE*, ktoré slúžia na volanie ostatných funkcií, ktoré obsahuje bytecode. Po vykonaní všetkých inštrukcií v zavolanej funkcií sa pokračuje na ďalšiu inštrukciu, ktorá nasleduje po inštrukcii *INVOKE*.

3.3.4 Výsledok uzlu

Po vykonaní všetkých inštrukcií sa ako výsledok uzla berie *XOR* hodnôt na zásobníku.

3.4 Problémy spojené s implementáciou

Jeden z problémov v súčasnej implementácií 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]. S týmto rozdelením sme schopní vykonať maximálne 255 inštrukcií, a zároveň začať maximálne na riadku 255. V bežnom prípade je teda posledná inštrukcia, ktorú dokážeme vykonať na riadku 510, čo znamená, že akákoľvek inštrukcia za ňou nemôže byť nikdy vykonaná. Avšak funkcie v bytecode majú často niekoľko násobne viac inštrukcií. 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 v rámci jedného uzlu veľmi rýchlo. Keďže môže nastať situácia, kedy sa v extrémnych prípadoch vykoná aj viac ako 255 inštrukcií, bolo potrebné limitovať celkový počet inštrukcií, ktoré sa môžu vykonať v rámci emulácie jedného uzlu, na 300. Situácia kedu sa môže vykonať viac ako 255 inštrukcií môže nastať napríklad kvôli inštrukcii *GOTO*, kedy sa stáva že simulátor donekonečna preskakuje niekam nad aktuálnu inštrukciu a toto opakuje až kým sa nevyčerpá limit inštrukcií. Alebo pri inštrukciách *INVOKE* v prípade, že sa zavolá funkcia, ktorá obsahuje priveľa inštrukcií.

3.5 Implementačné rozdiely medzi skutočným JVM a našim JVM

Okrem rozdielu, že nevykonávame funkcie v bytecode ako celok, ale pre každý uzol vykonávame náhodný kus inštrukcií z náhodnej funkcie, je ďalším rozdielom neúplnosť implementácie inštrukcií v našom JVM simulátore. Dôvodom je, že nie všetky inštrukcie boli pre nás výhodné na implementáciu čo sa týka pomeru vynaloženého úsilia a pridanej hodnoty. Preto sa pri vykonávaní týchto inštrukcií nevykoná nič a tieto inštrukcie sa automaticky preskočia. Jednými z nich sú napríklad inštrukcie na prácu s poľami a objektami. Celý zoznam inštrukcií, ktoré JVM simulátor síce obsahuje, ale nemá ich implementované, je vypísaný v prílohe.

3.6 Výhody prístupu

Okrem výhody používať vo funkcii, ktorá je vytváraná za účelom rozlíšiť testované dáta od náhodných 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 silný nástroj, 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.

3.7 Nevýhody JVM uzlov

Najväčšou nevýhodou je dĺžka výpočtu, avšak takáto dĺžka je očakávateľná, pretože sa v uzloch vykonávajú časovo o dosť zložitejšie výpočty pri porovnaní s bežnými uzlami. Zatiaľ čo bežný uzol sa dá prirovnať k jednej vykonanej inštrukcii medzi všetkými vstupmi, JVM simulátor ich vykonáva v rámci jedného uzlu niekoľko násobne viac. S vykonávaním inštrukcií je spojená aj réžia, ako napríklad kontrola počtu hodnôt na zásobníku alebo kontrola prekročenia maximálneho počtu inštrukcií. Takže v konečnom dôsledku je jeden beh mnohonásobne dlhší, ako beh s 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 som preto musel vyskúšať viacero experimentov, ktoré si predstavíme v tejto kapitole, a porovnáme ich výsledky s výsledkami, ktoré dosiahol EACirc s bežnými uzlami pri rovnakých nastaveniach.

Autorom výsledkov z EACircu s bežnými uzlami nie som ja, ale pochádzajú z interných stránok EACircu.

4.1 Experiment s imitáciou bežných uzlov

Prvý experiment slúžil najmä 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 som musel 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.

Jednou z vecí, ktorú je zložité napodobniť v prípade JVM uzlov je, 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. Problémom je, že bežné uzly vykonajú operáciu vždy medzi všetkými vstupmi, zatiaľ čo JVM simulátor vykonáva v jednom uzle náhodné množstvo inštrukcií. Preto bolo potrebné premyslieť, ako ich vykonať dostatok na to, aby sa zvolená inštrukcia vykonala medzi všetkými vstupmi, teda medzi všetkými hodnotami na zásobníku. Tento problém by sa dal vyriešiť tým, že by sme vykonávali vždy všetky inštrukcie, ktoré funkcia obsahuje, avšak toto riešenie by by bolo v rozpore so základnou myšlienkou JVM uzlov, a tou je vykonávať náhodný kus inštrukcií, ktoré pochádzajú zo šifrovacej funkcie. Je zrejmé, že ak budeme vykonávať vždy náhodnú časť inštrukcií, nie v každom prípade dosiahneme požadovanú funkcionalitu. Zvolili sme preto možnosť, že pre funkcie, ktoré potrebujú spracovať všetky hodnoty zo zásobníka, aspoň zvýšime pravdepodobnosť, že sa vybraných inštrukcií vykoná dostatok. Preto každá takáto funkcia obsahuje viackrát pod

sebou vybranú inštrukciu tak, aby genetika mala pri náhodnom vyberaní inštrukcií väčšiu šancu zvoliť väčší počet inštrukcií. Inštrukciu sme vybrali podľa toho aby plnila rovnakú funkcionalitu, napríklad v prípade funkcie *AND* je to inštrukcia *iand*.

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. Podobne ako v predchádzajúcom prípade, aj tu sme nepoužili emulovanie vždy celej funkcie, kvôli rozporu so základnou myšlienkou JVM uzlov.

Ďalší problém, ktorého výskyt sa ešte znásobil pri použití funkcií, ktoré vykonávajú náhodné množstvo rovnakých inštrukcií, je, že tieto inštrukcie potrebujú na zásobníku minimálne toľko hodnôt, aký je ich počet. Preto sme sa museli zamyslieť aj nad otázkou, ako sa zachovať, ak je na zásobníku nedostatočný počet hodnôt. Inými slovami čo spraviť, ak potrebujeme vybrať hodnotu z prázdneho zásobníka. Z tohoto dôvodu som musel upraviť implementáciu s nasledujúcimi možnosťami.

- **Pri vyberaní hodnoty z prázdneho zásobníka vracať 0**
  Toto riešenie sa ukázalo ako nesprávne, napríklad v súvislosti s inštrukciou *AND*. Súčasťou vykonania inštrukcie AND je výber dvoch hodnôt zo zásobníka. Avšak, vo chvíli keď sa na zásobníku nachádza len jedna hodnota, simulátor vyberie práve túto jednu hodnotu a za druhú, neexistujúcu hodnotu, dosadí vždy nulu. A keďže výpočet *AND-u*, v ktorom sa nachádza 0 je vždy 0, strácame celý doterajší výpočet.
- **V prípade prázdneho zásobníka vracať neutrálnu hodnotu**
  Tento spôsob by vyriešil problém predchádzajúceho riešenia, pretože v prípade, keď sa vyberá z prázdneho zásobníka vracala by sa taká hodnota, s použitím ktorej by sa operácia správala ako identita, vrátila by 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 sme zvolili ako najlepšie z dvoch dôvodov. Prvý sa týka výkonu, nakoľko sa nepočítajú zbytočné výpočty a druhý sa týka vyriešenia pôvodného problému.

4.1.2 Výsledky experimentu

Takto vytvorený bytecode sme použili na rozlíšenie medzi náhodnými dátami a hašovacou funkciou Tangle. 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 som sa rozhodol, že takto budem označovať len bunky, ktoré budú mať hodnotu nad 0.07.

Z výsledkov v tabuľke 4.1 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

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.

bežné uzly boli schopné v 94% behov nájsť hľadaný obvod, EACirc s JVM uzlami bol to isté schopný spraviť len v 27% behov. Dôvodom menšej úspešnosti sú zrejme nevyriešené problémy spomenuté vyššie.

Pri výpočte na 300000 generáciách bolo EACircu poskytnuté 10 krát viac generácií na hľadanie obvodu, čiže sú očakávané lepšie výsledky. Avšak ani v tomto prípade nedosiahol EACirc s JVM uzlami lepšie výsledky ako mali bežné uzly. Preto som sa rozhodol pokračovať v testovaní ďalšieho experimentu, v ktorom už budú použité bytecody zo šifrovacích respektíve hašovacích funkcií.

4.2 Experiment s bytecodom z kandidátnych funkcií SHA3 a eStream

V tomto experimente som plnohodnotne využil najväčšiu výhodu JVM uzlov a to použitie bytecodu z reálnych funkcií, konkrétne z hašovacích funkcií zo sútaže na funkciu SHA-3: *Tangle, Dynamic SHA, Dynamic SHA-2* a z prúdovej šifrovacej funkcie z kandidátov na funkciu eStream: *Decim*. Experiment spočíval v tom, že som z týchto všetkých funkcií vytvoril bytecode a spustil 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 mi nepodarilo nájsť implementáciu ani jednej z týchto funkcií v Jave, preto som musel vytvoriť vlastnú. Najjednoduchšie riešenie, na ktoré som prišiel, bolo prepísať implementáciu z C++, ktorá je dostupná napríklad aj priamo v EACircu, do Javy. K prepisu mi pomohla voľná verzia konvertoru [Tan] z C++ do Javy, od spoločnosti *Tangible Software Solutions Inc.* Po konvertovaní som musel 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. Implementácia,

ktorá vznikla týmito krokmi by zrejme nebola plne funkčná ani korektná, ale na naše účely, teda na využitie inštrukcií, je postačujúca.

Takto vytvorenú implementáciu som potom jednoducho skompiloval a zo skompilovaného súboru získal konkrétne inštrukcie. Vzniknutý súbor s bytecodom bolo treba jemne upraviť, aby si s ním poradila funkcia na načítavanie bytecodu, ktorú obsahuje JVM simulátor. Išlo napríklad o vymazanie niektorých tabulátorov a podobne. Posledným krokom bolo implementovanie inštrukcií, ktoré JVM simulátor zatiaľ nepoznal, pretože inak by načítavanie zlyhalo. Nie v každom prípade bolo potrebné inštrukcie aj implementovať (viac sekcii 3.5).

4.4 Výsledky experimentu

Pre každú funkciu, som spustil výpočty pre 3 rundy. Rundy som vyberal podľa úspešnosti bežných uzlov tak, že som našiel poslednú rundu, kde sme označili EACirc s bežnými uzlami za úspešný a do mojich výpočtov zahrnul túto rundu plus predchádzajúcu a nasledujúcu. Pre každú rundu som spustil výpočty v kombinácií s každým bytecodom, dohromady teda 4 výpočty na každú rundu, pretože máme 4 bytecody. To znamená, že dohromady som spustil 12 rôznych výpočtov pre každú funkciu. Každý z 12-tich výpočtov sme spustili 1000 krát, čo znamená, že dovedna som potreboval vypočítať pre jednu funkciu 12000 behov EACircu. Výpočty som spúšťal na metacentre [Tea16], kde sa behy spúšťajú v rámci tzv. úloh. V rámci jednej úlohy zvládlo metacentrum vypočítať 8 behov za menej ako dva dni. Dĺžka výpočtu závisela od stroja, na ktorom sa úloha počítala. Niektoré zvládli všetkých 8 behov za 10 hodín, zatiaľ čo niektorým to trvalo aj 30 hodín. Keďže metacentrum naraz priraďuje až 800 virtuálnych strojov s jedným procesorom, všetky úlohy pre jednu funkciu sa stihli vypočítať za približne 2 až 3 dni.

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

Výsledky pre funkciu Tangle					
Runda	**Bytecode z funkcie**				
	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, dokonca aj s bežnými uzlami, je schopný rozlíšiť všetky rundy funkcie *Dynamic SHA*, zatiaľ čo štatistické sady (kapitola 1.)

Výsledky pre funkciu Dynamic SHA					
Runda	**Bytecode z funkcie**				
	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.

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 a funkcia buď nespĺňa niektorú z požiadaviek na náhodnosť, ktorú štatistické testy netestujú, alebo sa v implementácii EACircu nachádza chyba, ktorú sme prehliadli.

Výsledky pre funkciu Dynamic SHA-2					
Runda	**Bytecode z funkcie**				
	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.

Výsledky z tabuliek 4.2 až 4.5, sú trochu nejednoznačné. Z môjho pohľadu som v nich nenašiel 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 na rozdiel od experimentu 1 dosiahli JVM uzly vo väčšine prípadoch porovnateľné výsledky ako bežné uzly. Najhoršie výsledky sme dosiahli v prípade funkcie Decim, preto sme sa rozhodli, že na tejto funkcii vyskúšame výpočty s viac generáciami. Počet generácií sme zvolili na 300000, tým pádom bude každý beh EACircu vylepšovať výsledný obvod 10-krát dlhšie. Z dôvodu veľmi dlhého výpočtu, sme museli znížiť počet behov z 1000 na 400.

Tabuľka 4.6 neobsahuje výsledky pre bežné uzly, pretože výsledky pre bežné uzly s 300000 generáciami nemáme k dispozícii. Výsledky sa dajú porovnať s výsledkami z tabuľky 4.5, konkrétne stĺpec s označením bežné uzly. Treba však brať na vedomie, že bežné uzly hľadali obvod 10-krát menej generácií. Čo sa týka výsledkov, nie je v nich nič prekvapivé, zase sme sa trochu priblížili k výsledkom, ktoré dosiahol EACirc s bežnými uzlami na 30000 generáciách.

Výsledky pre funkciu Decim					
Runda	**Bytecode z funkcie**				
	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	1.000	0.954	0.967	0.659
5	0.669	0.453	0.465	0.320
6	0.039	0.061	0.048	0.058

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

4.5 Výpočty so zložitejším bytecodom

Ďalší a zároveň posledný experiment, ktorý som vyskúšal, bol postavený na myšlienke použitia bytecodu, 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 som pre funkciu Decim spustil výpočty aj pre verziu s 300000 generáciami a 400 behmi.

Z tabuliek 4.7 až 4.10 vyplýva, že EACirc s bytecodom z funkcie AES sa úspešnosťou vyrovnáva použitiu bytecodu z funkcií v predchádzajúcom experimente až na funkciu Decim, kde dosiahol výrazne lepších výsledkov, 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.

Vo výsledkoch v tabuľke 4.11 sa podarilo JVM uzlom prekonať EACirc s bežnými uzlami. Avšak bežné uzly boli spúštané len s 30000 generáciami, takže mali na hľadanie výsledného obvodu mali 10 krát menej generácií. Dokonca sa nám podarilo prekonať bežné uzly o jednu rundu, aj keď iba veľmi tesne a je otázne, či sa náhodou nejedná len o štatistickú odchýlku.

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	1.000
5	0.982
6	0.072

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).