Masarykova univerzita
Fakulta informatiky

Rozšírenie frameworku EACirc o simulátor Java bytecodu

Bakalárska práca

Michal Hajas

Brno, jar 2016

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.

Michal Hajas

Vedúci práce: Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D

Poďakovanie

Týmto by som sa chcel poďakovať Ing. Mgr. et Mgr. Zdeňkovi Říhovi, Ph.D., za vedenie bakalárskej práce a čas strávený podávaním rád a pripomienok. Takisto by som sa chcel poďakovať celému tímu, ktorý pracuje na projekte EACirc, ale aj laboratóriu CRoCS na Fakulte informatiky Masarykovej univerzity za možnosť zapojiť sa do vylepšovania tohto nástroja a za množstvo nových vedomostí, ktoré som sa pri spolupráci s nimi naučil.

Oceňujem aj možnosť pristupovať k výpočtovým a úložným zdrojom, ktoré patria Národnej Gridovej Infraštruktúre MetaCentrum a sú poskytované pod programom "Projects of Large Research, Development, and Innovations Infrastructures"(CESNET LM2015042).

V neposlednom rade ďakujem aj svojim rodičom, sestre, priateľke a všetkým kamarátom za to, že ma počas celého štúdia podporovali a pomáhali mi nielen pri úprave tejto práce.

Zhrnutie

Táto práca ukazuje rôzne možnosti ako testovať náhodnosť a predstavuje nový nástroj ako alternatívu k nástrojom, ktoré sa na testovanie náhodnosti bežné používajú. Jej cieľom je vytvoriť a popísať rozšírenie frameworku EACirc o simulátor Java bytecodu, teda upraviť jeho implementáciu tak, aby počas behu využíval inštrukcie, ktoré pochádzajú z programu, ktorý je napísaný v jazyku Java. Vytvorená implementácia je ďalej otestovaná a výsledky sú porovnané s výsledkami, ktoré dosiahla implementácia bez tohto rozšírenia.

Abstract

This thesis presents several ways how to test randomness and introduce new tool as an alternative to tools which are mostly used for testing randomness. The primary goal of this thesis is to implement and describe simulator of Java bytecode as an extension of framework EACirc. Which means to enhance implementation of framework EACirc so that it would be possible to use instructions, which comes from a programme, that is programmed in Java language. Created implementation is tested and the results are compared to results, which was achieved by implementation without this extension.

Kľúčové slová

bytecode, CRoCS, EACirc, Java virtual machine, JVM, testovanie náhodnosti, inštrukcie, štatistické testy, genetické programovanie, samovzdelávací algoritmus

Obsah

Úvod

Použitie kryptografie sa stalo v dnešnej dobe nevyhnutnou súčasťou bežného života. Každé spojenie, cez ktoré komunikujeme je nejakým spôsobom zabezpečené práve pomocou kryptografických funkcií. Aj najmenšia chyba v takýchto funkciách by mohla spôsobiť zneužitie súkromných dát a tomu sa snaží množstvo odborníkov na celom svete zabrániť. Preto potrebujeme stále lepšie nástroje, ktoré dokážu odhaľovať nedostatky implementácie kryptografických funkcií.

Jednou z požiadavok na kryptografické funkcie je aj nemožnosť zistiť, že dáta pochádzajú z kryptografickej funkcie, pretože potencionálnemu útočníkovi by mohlo takéto zistenie pomôcť v prelomení zašifrovaných dát. Aby sa nedalo určiť, či dáta pochádzajú z kryptografickej funkcie, mali by sa tieto dáta čo najviac podobať na náhodnú sekvenciu. Z tohto dôvodu vznikli nástroje, ktoré zisťujú pravdepodobnosť, že sú skúmané dáta náhodné. V prvej kapitole si predstavíme nástroje, ktoré používajú veľké korporácie, ktoré určujú kryptografické štandardy na testovanie tejto vlastnosti. Jedná sa o štatistické sady testov na testovanie náhodnosti. Predstavíme si 3 konkrétne najpoužívanejšie štatistické sady a síce STS NIST [Ruk+00], Dieharder [Bro04] a TestU01 [LS07].

Z dôvodu niektorých nedostatkov štatistických testov vznikol k štatistickým sadám alternatívny prístup, ktorý si predstavíme v druhej kapitole. Jedná sa o framework EACirc, ktorý používa na testovanie náhodnosti funkcie, ktoré rozlišujú medzi náhodnými dátami a výstupom z kryptografickej funkcie. Takáto funkcia je EACircom vytváraná automaticky. Na vytváranie sa používa samovzdelávací genetický algoritmus. Výhodou EACircu je, že tieto testy sa vytvárajú takmer bez potreby ľudského zásahu, zatiaľ čo testy zo štatistických sád sú väčšinou zložité funkcie, nad ktorými museli ľudia stráviť množstvo času.

Táto práca bola vytvorená za účelom rozšíriť framework EACirc o simulátor Java bytecodu. V tretej kapitole si preto vysvetlíme čo je to Java bytecode a takisto ako sa takýto bytecode vykonáva pomocou vytvoreného JVM simulátora. Ďalej si ukážeme ako je tento JVM simulátor prepojený s implementáciou EACircu. Požiadavka na toto rozšírenie bola, aby sa v rámci EACircu mohli vykonávať náhodné kusy inštrukcií, ktoré pochádzajú z programu, ktorý je napísaný v Jave.

V poslednej kapitole nájdeme popis a výsledky experimentov, ktorými bola otestovaná implementácia EACircu s novým rozšírením a to s použitím JVM simulátora. Takisto tam nájdeme zhrnutie jednotlivých výsledkov a porovnanie s výsledkami zo štandardného EACircu. Postupne si predstavíme 3 experimenty.

- Prvý z nich bol zameraný na otestovanie funkčnosti implementácie EACircu, ktorá obsahuje JVM simulátor.
- Druhý experiment bol zameraný na zistenie toho, či má EACirc, ktorý v sebe používa inštrukcie z nejakej konkrétnej kryptografickej funkcie, väčšiu šancu na rozlíšenie výstupu z tejto funkcie ako EACirc s bežnými uzlami.

- A nakoniec posledný experiment, v ktorom sa budeme snažiť zistiť, či je genetika schopná poskladať hľadanú funkciu aj zo zložitejšieho bytecodu.

EACirc je tímový projekt, na ktorom sa pracuje v rámci laboratória CRoCS [Cen] (Centre for Research on Cryptography and Security) na Fakulte informatiky Masarykovej univerzity. Preto sú niektoré súčasti tejto práce výsledkom tímovej spolupráce. Úpravy implementácie a experimenty boli realizované vo väčšej miere mnou, avšak po dohovore s ostatnými členmi tímu. Mojou úlohou bolo doplniť už existujúcu implementáciu JVM simulátora tak, aby sa dala táto implementácia používať v rámci frameworku EACirc. Taktiež som autorom všetkých výpočtov, ktoré testovali tento nový prístup.

Zdrojový kód frameworku EACirc je voľne dostupný na GitHube [Š+12]. Implementácia, ktorú vysvetľuje táto práca je dostupná vo vetve, ktorá ma názov jvmsim. Stránky obsahujú aj prehľadnú dokumentáciu, kde sa dá nájsť viac informácií v prípade nejasností.

Text práce bol vytvorený v sádzacom systéme LaTeX s využitím balíka *fithesis2* [FIL18].

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, ktorý je následne spolu s výsledkami ostatných testov skombinovaný. Keďže sa nikdy nedá určiť, či sú dáta náhodné alebo nie na 100 percent, 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 napríklad malá pravdepodobnosť, že naozaj náhodná sekvencia bude obsahovať len samé jednotky, naopak s oveľa vyššou pravdepodobnosťou bude počet 1 a 0 približne rovnaký. Z tohto dôvodu je jedným z testov, ktorý obsahujú štatistické sady aj test, ktorý testuje či je táto vlastnosť splnená. Niektoré z testov majú dokonca možnosť nastaviť parametre, vďaka ktorým sa môže nad dátami spustiť tento test vo viacerých variantách. To znamená, že konečný počet spustených testov môže byť niekoľko násobne väčší ako počet testov, ktoré sada obsahuje. Cez rozhranie na príkazovom riadku sa dá pred spustením nastaviť kombinácia testov podľa toho, aké výsledky chceme získať. 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 bude interval rozdelený na 10 častí podľa hodnoty, teda prvá časť bude obsahovať p-hodnoty z intervalu $[0, 0.1]$, druhá časť p-hodnoty z intervalu $[0.1, 0.2]$ atď. malo by v každej časti skončiť rovnaké množstvo p-hodnôt, v našom prípade približne 100.
- **Pomer úspešných testov ku všetkým testom**
  Ďalší spôsob na určovanie výsledku je pomer úspešných 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-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 počtu sekvencií, ktoré testami prešli, ku počtu všetkých testovaných sekvencií, 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 si na testovanie zvolili 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, čo sú v podstate skupiny testov, alebo samostatne. Testy sú implementované v jazyku ANSI C. Knižnica je rozdelená do viacerých modulov, ktorých

popis je k dispozícii v dokumentácii [LEc09] a 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 a 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 crush, aj crush slúži na testovanie generátorov.
- **Big crush**
  Ešte väčšia a pomalšia batéria ako crush a small crush.
- **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 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 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 pravdepodobnosť, že by takáto sekvencia bola výstupom z naozaj náhodného generátora je veľmi malá. Nevýhodou štatistických sád je to, že obsiahnuté testy 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* [Š+12].

Prístup EACircu je oproti štatistickým testom úplne odlišný. Vo svojej podstate je to nástroj, ktorému nastavíte dva prúdy dát a on hľadá 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. Inými slovami by sa dalo povedať, že EACirc dokáže byť v určovaní náhodnosti len taký dobrý, ako dobré su náhodné dáta, ktoré používa.

Častou požiadavkou na EACirc je to, aby fungoval podobne ako štatistické sady, teda aby rozlišoval medzi náhodnými dátami a dátami pochádzajúcimi zo šifrovacej respektíve hašovacej funkcie. Preto EACirc obsahuje integrované kryptografické funkcie, ktoré sa dajú jednoducho nastaviť ako jeden z prúdov. Jedná sa napríklad o kandidátov na funkciu SHA3 [Dub12], kandidátov na funkciu eSTREAM [Pri12] alebo tiež kandidátov zo súťaže CAESAR [Ukr16]. Výsledky obsahujúce aj porovnanie so štatistickými testami pre funkcie zo súťaží SHA3 a eSTREAM, sú k dispozícii v nasledujúcich publikáciách [Ukr13; SUM13; ŠUM14].

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 spĺňala 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 2.3), 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 2.2) vytvoril funkciu, ktorá bude vykonávať postupnosť týchto operácií a dokáže zo vstupu zistiť, či dostala náhodné dáta alebo preverované dáta. S týmto prístupom sa spája niekoľko výhod, napríklad:

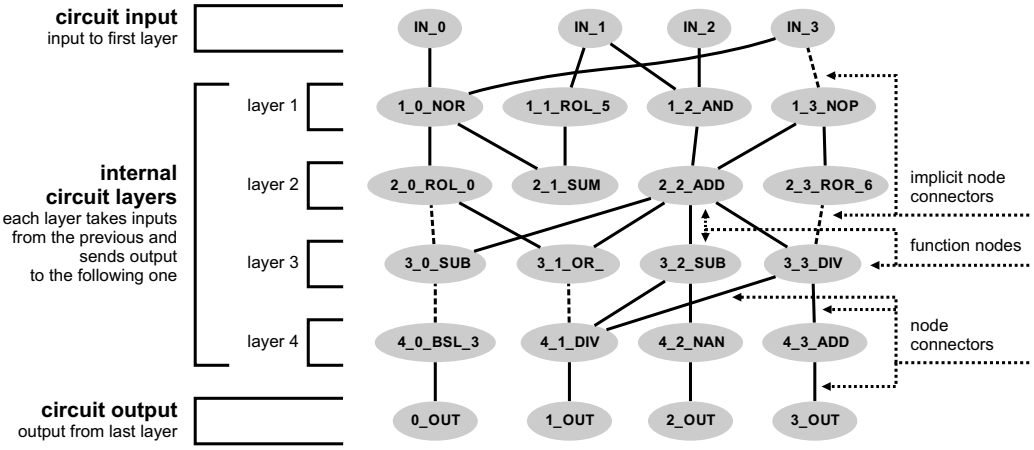
- **Na vytváranie testov nie je potrebná žiadna ľudská aktivita**
  Testy zo štatistických sád bývajú založené na matematických problémoch, nad ktorými museli ľudia stráviť množstvo času.
- **Testovanie aj zatiaľ nepoznaných problémov**
  Štatistické sady neobsahujú testy na všetky vlastnosti nenáhodných dát, na druhej strane EACirc vytvára hľadanú funkciu náhodne, z toho vyplýva, že v nej môže teoreticky testovať čokoľvek, k čomu genetický algoritmus dospeje.

2.2 Vysvetlenie funkcie na rozlišovanie prúdov

Celá skladačka je v skutočnosti graf. Každá kocka tejto skladačky reprezentuje jeden uzol a spojenie dvoch kociek znamená, že medzi týmito uzlami vedie cesta. Tento graf je zoskupený do horizontálnych vrstiev, ktoré sú poskladané na seba. V každej vrstve sa nachádza niekoľko uzlov. Cesty vedú len smerom zhora nadol a to len medzi vrstvami idúcimi bezprostredne za sebou tak, ako to je vidieť na obrázku 2.1. Prvá vrstva je vstupná a posledná výstupná. V ďalších odsekoch si vysvetlíme princíp fungovania grafov, ktoré budeme nazývať aj obvody. Teda akým spôsobom dokáže jednoduchý graf určiť, či dostal dáta z prvého, respektíve druhého prúdu.

Každý uzol v sebe nesie uloženú informáciu. Na tomto mieste je okrem iného uložené aj to, akú funkciu bude tento uzol vykonávať. Každý z uzlov môže vykonávať práve jednu z nasledujúcich funkcií:

- **Bitové operátory**
  AND, OR, XOR, NOR, NAND, ROTL, ROTR, BITSELECTOR



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

- **Aritmetické funkcie**
  SUM, SUBS, ADD, MULT, DIV
- **Operátor identity**
  NOP
- **Operátor na priame čítanie zo vstupu**
  READX

Uzol v sebe ďalej môže obsahovať aj parametre, ktoré niektoré funkcie potrebujú na ich vykonanie. Celkový limit na uložené dáta v rámci jedného uzlu je 4B.

Vykonávanie funkcie v rámci jedného uzla prebieha nasledovne. Na začiatku uzol dostane niekoľko vstupov z predchádzajúcej vrstvy. Všetky tieto vstupy spracuje na základe funkcie, ktorú ma vykonávať, napríklad vykoná medzi všetkými vstupmi operáciu *AND*. Konečný výsledok posiela uzlom v nasledujúcej vrstve ako vstup. Voľba od koho uzol dostane dáta a kam ich po vykonaní pošle prebieha náhodne. Preto môže nastať aj situácia, kedy nemá uzol žiadne vstupy ani výstupy, to znamená, že takýto uzol vôbec neovplyvňuje konečný výsledok. Prvá vrstva je vstupná, čiže na vstupe dostane dáta priamo z niektorého z prúdov. Následne dáta prebublávajú smerom nadol tak, že každý uzol z dátami vykoná funkciu, ktorú ma v sebe uloženú a výsledok pošle o úroveň nižšie. Posledná vrstva je výstupná, to znamená, že z dát ktoré sú výstupom z poslednej vrstvy sa zisťuje, či dáta pochádzajú z prvého, alebo druhého prúdu.

Použitie jednoduchých funkcií môže v konečnom dôsledku vyústiť do zložitejších konštrukcií. Dokonca sa môže docieliť podobného testu ako sa vyskytuje v štatistických sadách. 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 vykonávať v uzloch niečo zložitejšie, napríklad inštrukcie pochádzajúce z kódu, ktorý vygeneroval testovaný prúd dát. V kapitole 3. si tieto uzly bližšie predstavíme.

2.3 Genetika

V tejto sekcii si objasníme prečo sa algoritmus používaný na vytváranie najlepšieho obvodu nazýva genetický. Ďalej si preberieme aj to, ako sa postupne z obyčajného náhodného grafu, ktorý nemá najlepšiu úspešnosť v rozlišovaní toho, z ktorého prúdu dostal na vstupe dáta, vytvorí graf, ktorý má spomenutú úspešnosť vyššiu.

Genetické programovanie [Koz92] vzniklo za účelom naučiť počítače riešiť rôzne problémy bez toho, aby na to boli explicitne naprogramované. Teda aby počítače vedeli riešiť problémy bez toho, aby im bolo povedané ako presne to urobiť. V skutočnosti sa jedná o algoritmy výsledkom ktorých je štruktúra, ktorá nejakým spôsobom reprezentuje počítačový program. V prípade EACircu je touto štruktúrou graf. Keďže sa jedná o genetický algoritmus, tento graf označujeme okrem pojmu obvod aj jedinec.

Algoritmus sa volá genetický aj preto, lebo jeho priebeh pripomína fungovanie genetiky v prírode. Priebeh algoritmu sa delí na tzv. generácie. V každej generácií existuje niekoľko jedincov. Každý jedinec je reprezentovaný grafom a cieľom je postupne tieto grafy vylepšovať tak, aby v každej nasledujúcej generácii boli úspešnejší jedinci v riešení zadaného problému, v našom prípade v riešení problému náhodnosti. Vytváraním nových generácií sa docieľuje toho, že sa jedinci postupne učia ako riešiť zadaný problém, preto sa tento algoritmus nazýva aj samovzdelávací [Ban98]. Celý priebeh algoritmu sa dá opísať nasledujúcimi krokmi:

1. **Náhodné vygenerovanie jedincov**
   Vytvorí sa tzv. iniciálna populácia jedincov. Tieto jedince sa tvoria náhodne, to znamená, že sa bude jednať o niekoľko grafov, kde každý bude vytvorený náhodne.

2. **Určenie úspešnosti**
   Úspešnosť sa vypočíta pomocou tzv. funkcie vhodnosti, ktorá je pre správny priebeh veľmi dôležitá. Každý z jedincov bude mať nejakú úspešnosť, niektorí ju budú mať lepšiu a niektorí horšiu.

3. **Vyradenie neúspešných obvodov**
   Keďže potrebujeme aby nasledujúca generácia bola čo najlepšia, musíme ju založiť len na tých najlepších jedincoch, slabšie jedince preto vyradíme.

4. **Skríženie najúspešnejších jedincov**
   Skrížením dostaneme novú populáciu jedincov. Cieľom kríženia je dosiahnuť novú silnejšiu generáciu.

5. **Náhodná mutácia niektorých jedincov**
   Aby sa zabránilo zaseknutiu v lokálnom maxime, je potrebné urobiť náhodnú mutáciu, napríklad odobrať niektorú z ciest, alebo zmeniť niektorý z uzlov. Mutácia prebieha tak, že sa postupne prechádza grafom 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.

Úspešnosť jedinca sa zistí veľmi jednoducho. Z prúdov dát sa získa niekoľko vzoriek, ktoré sa nazývajú testovacie vektory. Dĺžka testovacieho vektora sa dá zvoliť pre každý beh osobitne. Pri experimentoch v tejto práci som obvykle používal dĺžku 16B a počet vektorov 1000 z každého prúdu. Následne sa každému jedincovi predajú postupne všetky testovacie vektory a zistí sa s akou pravdepodobnosťou jedinec označil správne, či vektor pochádzal z prvého, respektíve druhého prúdu.

Po skončení sa zistí, či je výstupom jedinec, ktorý naozaj dokáže rozlíšiť, či sa jedná o dáta z prvého alebo druhého prúdu. Následne EACirc vyhlasuje, či sa jedná o náhodné dáta. Toto vyhlásenie je však relevantné len v prípade, že je jedným z prúdov prúd náhodných dát.

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 obvodov. V každom behu sa na začiatku zvolí náhodné počiatočné semienko a na ňom závisia všetky ďalšie vygenerované hodnoty. Takáto verzia generátora bola zvolená kvôli reprodukovateľnosti výpočtov. To znamená, kvôli tomu aby EACirc vracal rovnaké výsledky pre behy, ktoré majú rovnaké nastavenia a počiatočné semienko.

Na nastavenie všetkých hodnôt sa používa konfiguračný súbor, ktorý musí byť špecifikovaný pre každý beh. V rámci konfiguračného súboru sa nastavuje napríklad počet generácií, dĺžka testovacích vektorov alebo ich počet. Takisto sa v ňom volia prúdy, medzi ktorými sa bude v rámci behu rozlišovať. Príklad konfiguračného súboru je v prílohe.

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 preverovanými 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 sa počas celého priebehu algoritmu volia všetky hodnoty náhodne 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 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 pomer menší ako hladina významnosti, väčšinou 5%, znamená to, že EACirc bol úspešný v hľadaní obvodu

len vo veľmi málo prípadoch a teda skúmané dáta sú náhodné. Čím vyšší pomer, tým vo viac behoch bol EACirc úspešný v hľadaní obvodu a teda tým väčšia pravdepodobnosť, že dáta nie sú náhodné. Keďže sa spúšťa veľké množstvo behov EACircu naraz a v prípade veľkého množstva generácií spotrebuje jeden beh EACircu veľa času, bola do EACircu pridaná aj podpora zrýchlenia pomocou nVidia CUDA technológie [Nov15].

KAPITOLA 3

Simulátor Java bytecodu

V tejto kapitole si predstavíme nový typ uzlov ako náhradu za uzly predstavené v predchádzajúcej kapitole. Tieto uzly nevykonávajú iba jednoduchú funkcionalitu (napr. *AND, OR, XOR*), ale dokážu vykonávať inštrukcie získané z programu, ktorý je napísaný v jazyku Java. Celý princíp EACircu zostáva rovnaký, jedinou zmenou je vykonávaná funkcia v rámci jednotlivých uzlov. Nové uzly sa dajú jednoducho kombinovať s bežnými uzlami, pretože majú rovnakú štruktúru. To znamená, že v rámci jedného obvodu sa môžu použiť bežné uzly, a zároveň aj nové uzly. Tieto uzly vznikli za účelom pomôcť genetike, aby mohla čo najjednoduchšie vytvoriť výsledný obvod, ktorý bude mať čo najlepšiu úspešnosť v rozlišovaní medzi náhodnými dátami a preverovanými dátami. Základná myšlienka je taká, že obvod, ktorý v uzloch vykonáva inštrukcie, ktoré pochádzajú 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 inštrukcií z jazyka Java

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 programu, ktorý je napísaný v jave, získať nízko úrovňový binárny kód, ktorý sa dá interpretovať. Vďaka jej rozšírenosti by mala byť samozrejmosťou dostupnosť implementácie množstva pseudo náhodných generátorov a šifrovacích, respektíve hašovacích funkcií. Z tohto pohľadu sa Java javí ako veľmi dobrý jazyk pre jednoduché získavanie inštrukcií a následné vykonávanie v rámci uzlov EACircu.

3.2 Java Virtual Machine

JVM [MD97] je software, ktorý spája kód naprogramovaný v Jave a hardware konkrétneho počítača, na ktorom je tento kód spúšťaný. 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. Avšak narozdiel od hardvérového procesora, je JVM iba softvérové riešenie, ktoré ho napodobňuje.

Fungovanie JVM nie je nič zložité, obsahuje implementáciu všetkých inštrukcií a po načítaní súboru z inštrukciami, ktorý sa nazýva bytecode, vykonáva jednu inštrukciu po druhej. Avšak poznať detailné fungovanie JVM nie je pre účely tejto práce podstatné. Dôležité je vedieť, že aj náš JVM simulátor vykonáva inštrukcie z bytecodu, avšak s tým rozdielom, že vykonáva náhodný kus inštrukcií a vykonáva ich v rámci uzlov EACircu. Naša implementácia 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 a lokálne premenné, zatiaľ čo náš simulátor používa zásobník ako globálne úložisko, na ktoré sa ukladajú hodnoty počas celého jedného výpočtu, ktorý prislúcha 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 jej číslo, názov, prípadne argumenty, ktoré počas vykonávania využije. Napríklad inštrukcia *bipush* očakáva ako argument číslo, ktoré vloží na zásobník. Príklady bytecodu sú k dispozícii v prílohe.

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 rovnako ako pri bežných uzloch. V tejto sekcii si prejdeme celý proces krok po kroku, od načítavania bytecodu až po vykonávanie konkrétnych inštrukcií a následný 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 tohto 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 chyba, napríklad neznáma inštrukcia alebo zlá štruktúra bytecodu, vypíše sa chybová hláška 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. JVM uzol využíva celé 4 Byty a to nasledovne:

- 1. Byte: tu je uložená informácia, že sa jedná o JVM uzol. Každý typ uzla ma svoju konštantu, ktorá určuje aká funkcia sa má v rámci uzla vykonať. Napríklad ak je na tejto pozícii hodnota 19, znamená to, že sa jedná o JVM uzol.
- 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 riadka z parametra číslo 3 po inštrukciu na riadku, ktorý vznikne spočítaním 3. a 4. parametra. Toto obmedzenie však neplatí pri vykonávaní funkcie zavolanej špeciálnymi inštrukciami *invoke*. Význam týchto inštrukcií si rozoberieme v podsekcii 3.3.3.

JVM simulátor predpokladá, že každý uzol, ktorý bude vykonávať 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. Preto bola do EACircu doplnená funkcia, ktorá sa volá v prípade, že voľba vyberie, že sa jedná o JVM uzol. Táto funkcia vyberá parametre náhodne, avšak berie pri tom ohľad na to, aby bol výsledný uzol validný.

3.3.3 Vykonávanie inštrukcií

Ak EACirc pri vykonávaní obvodu narazí na JVM uzol automaticky zavolá JVM simulátor. Pred zavolaním funkcie, ktorá vykonáva inštrukcie, sa najprv musia predať JVM simulátoru všetky vstupy. Predávanie vstupov prebieha tak, že sa všetky vstupné hodnoty, ktoré idú do uzla, vložia na zásobník, ktorý obsahuje JVM simulátor. Až následne sa zavolá funkcia, ktorá sa stará o spúšťanie správnych inštrukcií. EACirc jej preto predá všetky parametre, ktoré sa v uzle nachádzajú, teda číslo funkcie, začiatočný riadok a počet inštrukcií, ktoré má vykonať. Okrem zásobníka obsahuje simulátor aj štruktúru, ktorá určuje stav procesora, teda obsahuje informácie o tom, ktorá funkcia sa vykonáva a na ktorom riadku. Po zavolaní funkcie sa táto štruktúra naplní dátami z uzla a následne sa pomocou nej posúva na každú nasledujúcu inštrukciu, ktorá sa bude vykonávať.

Existujú však aj inštrukcie, po vykonaní ktorých sa nepokračuje bežnou cestou. Teda pokračovaním na inštrukciu, ktorá nasleduje bezprostredne za práve vykonávanou inštrukciou. K týmto inštrukciám patria napríklad inštrukcie začínajúce na *if* teda napríklad: *ifeq, ifne, ifge* alebo *ifle*. Tieto inštrukcie slúžia na vetvenie programu, ich funkciou je overiť, či je splnená. V prípade, že simulátor narazí na takúto inštrukciu a zároveň je splnená kontrolovaná podmienka preskočí na konkrétnu inštrukciu, ktorá sa vyskytuje v rovnakej funkcii aká sa práve vykonáva. Číslo inštrukcie, na ktorú sa preskakuje 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, ktorých názov začína na *invoke*. Tieto slúžia na volanie inštrukcií z iných funkcií, ktoré obsahuje bytecode. Medzi inštrukcie *invoke* patria: *invokespecial, invokestatic, invokevirtual*. Po vykonaní všetkých inštrukcií v zavolanej funkcií sa pokračuje na ďalšiu inštrukciu, ktorá nasleduje po inštrukcii *invoke*. Teda napríklad ak sa pri vykonávaní funkcie číslo 1 zavolá funkcia číslo 2 pomocou niektorej z inštrukcií *invoke*, simulátor začne vykonávať všetky inštrukcie z funkcie číslo 2 a po ich vykonaní sa vráti naspäť do funkcie číslo 1 a pokračuje inštrukciou, ktorá nasleduje za inštrukciou *invoke*.

3.3.4 Výsledok uzlu

Cieľom vykonávania inštrukcií je spracovať nejakým spôsobom všetky vstupy, teda hodnoty na zásobníku a vytvoriť z nich hodnotu, ktorá sa predá na výstup. Keďže sa vykonáva náhodný kus inštrukcií, nemôžeme sa spoliehať na to, že bude po ich vykonaní na zásobníku len jedna hodnota. Preto sa po vykonaní všetkých inštrukcií ako výsledok uzla berie *XOR* všetkých hodnôt na zásobníku.

3.4 Problémy spojené s implementáciou

Jeden z problémov, ktorý má JVM simulátor je, že v súčasnej implementácií nie je možné uložiť do uzlu viac ako 4 Byty informácie. Keďže prvý Byte je rezervovaný pre funkciu, pre účely JVM simulátora ostávajú len 3 Byty. Po rozdelení máme teda pre každý z troch parametrov rozsah [0-255], teda 1 Byte. Z toho vyplýva, že s týmto rozdelením dokážeme vykonať maximálne 255 inštrukcií, a zároveň začať maximálne na riadku 255. V bežnom prípade, teda keď neberieme do úvahy inštrukcie, ktoré preskakujú na inú než nasledujúcu inštrukciu, je posledná inštrukcia, ktorú dokážeme vykonať, na riadku 510. Čo znamená, že akákoľvek inštrukcia za ňou nemôže byť vykonaná. Avšak funkcie v bytecode môžu mať niekoľko násobne viac inštrukcií. V súčasnej dobe však celá genetika v rámci EACircu prechádza zmenami a v novej implementácii by mal tento problém zaniknúť, pretože by malo byť v uzle viac miesta pre parametre.

Keďže existujú inštrukcie vďaka ktorým sa nemusí pokračovať na bezprostredne nasledujúcu inštrukciu, ale môže sa preskočiť kamkoľvek v rámci vykonávanej funkcie, môže sa stať, že sa vykonávanie zacyklí. Napríklad v prípade, že sa pomocou inštrukcie *goto* skáče niekam nad aktuálne vykonávanú inštrukciu. Následne sa na túto inštrukciu príde znova a znova sa preskočí na pôvodné miesto a toto sa opakuje donekonečna. Takáto situácia môže nastať kedykoľvek, najmä kvôli tomu, že vykonávame náhodný kus inštrukcií. Napríklad môže nastať situácia, kedy inštrukcia potrebuje niečo, čo poskytoval kód, ktorý sme preskočili. Preto sme potrebovali zamedziť tomu, aby takýmto spôsobom program pokračoval donekonečna. Ako riešenie postačilo limitovať počet vykonaných inštrukcií v rámci behu, ktorý prislúcha jednému uzlu. Tento limit bol nastavený na 300 inštrukcií.

Okrem pokračovania donekonečna môže nastať aj situácia, kedy sa program síce nezacyklí donekonečna, ale počet vykonávaných inštrukcií výrazne narastie. Táto situácia

môže nastať napríklad kvôli inštrukciám *invoke*, ktoré volajú iné funkcie v rámci bytecodu. Napríklad ak sa zavolá funkcia, ktorá má mnohonásobne viac inštrukcií ako 255, čo je maximum vykonaných inštrukcií v bežnom prípade. Keďže EACirc v rámci jedného behu vykonáva veľké množstvo uzlov, zavolanie takejto funkcie by mohlo nadmerne zvýšiť časovú náročnosť celkového behu EACircu. Preto aj v tomto prípade bolo potrebné nastaviť limit na 300 inštrukcií, v opačnom prípade by sa síce behy nepočítali donekonečna, ale medzi časmi jednotlivých behov by mohol byť výrazný rozdiel.

3.5 Implementačné rozdiely medzi skutočným JVM a našim JVM simulátorom

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. Okrem inštrukcií, ktoré JVM simulátor nepozná vôbec, existujú aj inštrukcie, ktoré síce pozná, ale nemá ich implementované. 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. Patria sem napríklad inštrukcie na prácu s poľami a objektami.

3.6 Výhody prístupu

Najväčšou výhodou je možnosť používať inštrukcie priamo z generátora, ktorý vytvoril preverované dáta. Okrem toho je ďalšou výhodou aj to, že bytecode sa môže skladať zo zložitejších častí a vďaka tomu je možné vytvoriť komplexnejšie obvody. Otázkou ale je, či je genetika natoľko silný nástroj, aby bola schopná zložiť takéto komplexné obvody, pretože s použitím JVM simulátora je 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. Ako odpoveď na tieto otázky môžu slúžiť výsledky experimentov, ktoré obsahuje táto práca.

3.7 Nevýhody JVM uzlov

Najväčšou nevýhodou je dĺžka výpočtu, avšak takáto dĺžka je očakávaná, 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 môže v rámci jedného uzlu vykonať až 300. 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 tohto 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 v niektorých uzloch sa nevykoná vôbec nič. V konečnom dôsledku aj obvody, v ktorých sa nachádza veľké množstvo takýchto uzlov, môžu mať 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.

Pri porovnávaní s bežnými uzlami sú používané výsledky z Baseline experimentu [16].

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. Teda spracujú všetky vstupy, ktoré takýto uzol dostane. 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, pretože by sme mohli presne určiť, koľko inštrukcií sa má vykonať. Avšak toto riešenie 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 presne taký počet inštrukcií, aký je treba na spracovanie všetkých hodnôt na zásobníku. Inštrukciu sme vybrali podľa toho aby plnila rovnakú funkcionalitu, napríklad v prípade funkcie *AND* je to inštrukcia *iand*, ktorá vyberie zo zásobníka dve čísla, vykoná medzi nimi operáciu *AND* a výsledok vloží naspať na zásobník.

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, ani tu sme nepoužili vykonávanie 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. V opačnom prípade sa snažia vyberať hodnoty z prázdneho zásobníka. Preto sme sa museli zamyslieť aj nad otázkou, ako sa zachovať, ak je na zásobníku nedostatočný počet hodnôt. Z tohto dôvodu som musel upraviť implementáciu tak, aby vyberanie hodnôt z prázdneho zásobníka nebol problém a nijak neovplyvňoval doterajší výpočet. Do úvahy pripadali možnosti z nasledujúceho zoznamu.

- **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 výsledok výpočtu *AND-u*, v ktorom sa nachádza 0 je vždy 0, tak stratíme celý doterajší výpočet vždy, keď takáto situácia nastane.
- **V prípade prázdneho zásobníka vracať neutrálnu hodnotu**
  Tento spôsob by vyriešil problém predchádzajúceho riešenia, pretože v prípade, keď sa vyberá z prázdneho zásobníka by sa vracala taká hodnota, s použitím ktorej by sa operácia správala ako identita, teda by vracala 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, keďže sa obsah zásobníka nezmení.
- **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 som použili na rozlíšenie medzi náhodnými dátami a hašovacou funkciou Tangle. Pre každý výpočet som spustil 1000 behov na jeden výpočet, teda výsledok

v každej bunke je vypočítaný z 1000 behov a určuje percentuálny počet behov, ktoré boli úspešné. Výsledky boli vyhodnotené pomocou nástroja Oneclick [Obr15]. 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 0.055 alebo 0.06, 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.

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 JVM uzlov, ktoré napodobňujú bežné uzly.

Počet rúnd odráža silu kryptografickej funkcie. Čím viac rúnd, tým by mala byť funkcia silnejšia a dáta by mali vypadať náhodnejšie. Preto je očakávané, že pri niektorej runde už nebude EACirc úspešný v rozoznávaní medzi náhodnými dátami a výstupom z hašovacej funkcie Tangle. Rundy pre tento experiment som volil podľa výsledkov EACircu s bežnými uzlami, vybral som poslednú rundu kde je EACirc s bežnými uzlami úspešný a pridal som ešte predchádzajúcu a nasledujúcu.

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 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 kryptografickej 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 bolo prepísať implementáciu z C++, ktorá je dostupná napríklad aj priamo v EACircu, do Javy. K prepisu som použil voľnú verziu konvertora [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.

4.4 Výsledky experimentu

Pre každú funkciu, som spustil výpočty pre 3 rundy. Rundy som vyberal rovnako ako v predchádzajúcom experimente. Našiel som poslednú rundu, kde bol EACirc s bežnými uzlami 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 k dispozícii 4 bytecody. To znamená, že dohromady som spustil 12 rôznych výpočtov pre každú funkciu. Každý z 12-tich výpočtov som spustil 1000 krát, čo znamená, že dovedna som potreboval spustiť pre jednu funkciu 12000 behov EACircu. Výpočty som spúštal 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 jednej úlohy závisela od stroja, na ktorom sa 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.

Z každého prúdu som použil 1000 testovacích vektorov. 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% prípadoch, plus štatistická odchýlka. Výsledky som vyhodnocoval pomocou nástroja Oneclick.

Výsledky pre funkciu Tangle					
Runda	**Bytecode z funkcie**				
	Bežné uzly (podiel úspešných)	**Tangle** (podiel úspešných)	**Dynamic SHA** (podiel úspešných)	**Dynamic SHA-2** (podiel úspešných)	**Decim** (podiel úspešných)
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.

Výsledky pre funkciu Dynamic SHA					
Runda	**Bytecode z funkcie**				
	Bežné uzly (podiel úspešných)	**Tangle** (podiel úspešných)	**Dynamic SHA** (podiel úspešných)	**Dynamic SHA-2** (podiel úspešných)	**Decim** (podiel úspešných)
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 z tabuľky 4.3 sú trochu prekvapivé, pretože EACirc je schopný rozlíšiť všetky rundy funkcie *Dynamic SHA*, zatiaľ čo štatistické sady sú schopné rozoznať funkciu len po rundu 7. To isté platí aj pre EACirc s bežnými uzlami. 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 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 pre všetky funkcie. Na druhú stranu, výsledky v rámci funkcií sú celkom konzistentné. Pretože napríklad na funkcii *Decim* vidno, že pre bytecody z funkcií *Dynamic SHA* a *Dynamic SHA-2* má veľmi podobné výsledky. Čo je očakávateľné, pretože implementácia týchto dvoch funkcií je veľmi podobná a teda ich bytecody obsahujú podobné inštrukcie.

Ďalšie pozitívum je, že na rozdiel od prvého experimentu dosiahli JVM uzly vo väčšine prípadoch porovnateľné výsledky ako bežné uzly. Najhoršie výsledky sme dosiahli v prípade

Výsledky pre funkciu Dynamic SHA-2					
Runda	**Bytecode z funkcie**				
	Bežné uzly (podiel úspešných)	**Tangle** (podiel úspešných)	**Dynamic SHA** (podiel úspešných)	**Dynamic SHA-2** (podiel úspešných)	**Decim** (podiel úspešných)
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 pre funkciu Decim					
Runda	**Bytecode z funkcie**				
	Bežné uzly (podiel úspešných)	**Tangle** (podiel úspešných)	**Dynamic SHA** (podiel úspešných)	**Dynamic SHA-2** (podiel úspešných)	**Decim** (podiel úspešných)
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.

funkcie Decim, preto som sa rozhodol, že na tejto funkcii vyskúšam výpočty s viac generáciami. Počet generácií som zvolil na 300000, tým pádom bude každý beh EACircu vylepšovať výsledný obvod 10-krát dlhšie. Z dôvodu veľmi dlhého výpočtu, som musel znížiť počet behov z 1000 na 400. To znamená, že tieto výsledky by mali mať menšiu mieru významnosti oproti výpočtom, ktoré bežali až 1000 krát. Avšak pre vytvorenie obrazu o tom ako sa EACirc s JVM uzlami správa pri zvýšení počtu generácií by malo byť 400 behov dostačujúcich.

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 mali na hľadanie obvodu 10-krát menej generácií. Čo sa týka výsledkov, JVM uzly sa znovu priblížili výsledkom, ktoré dosiahol EACirc s bežnými uzlami na 30000 generáciách. Avšak neprekonal žiadnu ďalšiu rundu. Znovu si môžeme všimnúť mieru konzistencie v rámci tejto funkcie. Napríklad spojenie medzi výsledkami pre bytecode z funkcií *Tangle* a *Decim*. Je vidieť, že pre rundu 4 je výpočet s bytecodom z funkcie *Decim* približne o polovicu horší a to isté platí aj pre rundu 5.

Výsledky pre Decim s použitím 300000 generácií				
Runda	**Tangle** (podiel úspešných)	**Dynamic SHA** (podiel úspešných)	**Dynamic SHA-2** (podiel úspešných)	**Decim** (podiel úspešných)
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 úspešnejších obvodov. Na vytvorenie bytecodu sme použili implementáciu šifrovacej funkcie AES [01].

Nastavenia pre tento experiment boli rovnaké, ako v predchádzajúcom experimente, 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 výhodnejšie ako použitie bytecodu z jednoduchších funkcií.

Vo výsledkoch v tabuľke 4.11 sa podarilo JVM uzlom prekonať EACirc s bežnými uzlami. Avšak bežné uzly boli spúšťané len s 30000 generáciami, takže mali na hľadanie výsledného obvodu 10 krát menej generácií. Dokonca sa nám podarilo prekonať bežné uzly o jednu rundu, aj keď iba veľmi tesne. Keďže sa jedná o výpočet v ktorom bolo spustených len 400 behov, pravdepodobne sa bude jednať len o štatistickú odchýlku. Jediným spôsobom ako to overiť je spustiť ďalšie behy z tohto experimentu, avšak výpočet behov na 300000 generáciách je časovo veľmi náročný a preto som tieto výpočty už nespúšťal.

Tangle	
Runda	**Podiel úspešných**
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	**Podiel úspešných**
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	**Podiel úspešných**
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	**Podiel úspešných**
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	**Podiel úspešných**
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.

Záver

Bakalárska práca sa zaoberala rôznymi možnosťami testovania náhodnosti. Ukázali sme si niektoré nevýhody štatistických testov a predstavili sme nástroj EACirc, ktorý používa na testovanie náhodnosti úplne iný prístup ako štatistické testy.

EACirc používa na testovanie náhodnosti tzv. obvody. Tieto obvody sa vytvárajú automaticky a na ich vytváranie sa používa samovzdelávací genetický algoritmus. Ten funguje tak, že postupne, počas celého behu vylepšuje tento obvod tak, aby mal čo najlepšiu úspešnosť v určovaní náhodnosti. Tieto obvody sú v skutočnosti graf, ktorý obsahuje uzly a v konečnom dôsledku fungujú ako funkcia, ktorá rozlišuje medzi náhodnými dátami a výstupom z kryptografickej funkcie. Táto funkcia funguje tak, že uzlami grafu prechádzajú dáta a tie s nimi manipulujú a posielajú ich ďalším uzlom. Po tom čo dáta prejdú všetkými uzlami EACirc získa výsledok a vyhlasuje, či sú dáta náhodné.

Cieľom tejto práce bolo rozšíriť framework EACirc tak, aby bolo možné v rámci jeho uzlov vykonávať náhodný kus inštrukcií, ktoré pochádzajú z programu, ktorý je napísaný v jazyku Java. Tento cieľ bol splnený tým spôsobom, že sa vytvorili tzv. JVM uzly, ktoré sa pridali ako ďalší typ uzlov k už existujúcim. Tieto uzly sa v súčasnej implementácií dajú jednoducho použiť. Jedinou požiadavkou je špecifikovať v konfiguračnom súbore názov bytecodu a povoliť použitie JVM uzlov. Dokonca sa dajú použiť aj v kombinácií s bežnými uzlami, teda v rámci jedného obvodu sa použijú aj bežné uzly a zároveň aj JVM uzly.

Rozdiel medzi bežnými a JVM uzlami je taký, že bežné uzly vykonávajú len jednoduchú funkcionalitu, napríklad operáciu *AND* alebo podobné operácie, medzi všetkými vstupmi. Na druhej strane pri vykonávaní JVM uzla sa automaticky zavolá JVM simulátor, ktorý z uzla získa parametre a následne podľa nich vykoná náhodnú časť inštrukcií z dopredu načítaného bytecodu. Tieto inštrukcie manipulujú zo vstupmi a po vykonaní všetkých inštrukcií, dostávame výsledok, ktorý určuje či sú dáta náhodné alebo sú výstupom z kryptografickej funkcie. Tieto uzly majú nasledujúce výhody.

- Pomocou JVM uzlov je genetika schopná vytvoriť zložitejšie obvody.
- Na odlíšenie výstupu z kryptografickej funkcie od náhodných dát môžeme používať bytecode získaný priamo z tejto funkcie.

Avšak s týmto prístupom sa spájajú aj niektoré nevýhody. Napríklad:

- jednou z nich je dĺžka výpočtu, ktorá je oproti bežným uzlom mnohonásobne dlhšia.
- Je zložité implementovať celú funkcionalitu Java virtual machine, pretože obsahuje množstvo inštrukcií a prácu z množstvom dátových typov. Keďže v našom JVM simulátore nepodporujeme niektoré dátové typy, napríklad polia alebo objekty, tak sa niektoré inštrukcie automaticky preskakujú.
- Okrem toho, že je možnosť vytvorenia zložitých obvodov výhoda, môže to byť v niektorých prípadoch aj nevýhoda. Dôvodom je to, že keďže sa vytvárajú zložitejšie

obvody, aj genetika potrebuje vykonať zložitejšiu činnosť aby takýto zložitý obvod vylepšila a teda sa jej to nemusí vždy podariť.

Okrem rozšírenia EACircu o JVM simulátor bolo mojou úlohou aj vykonať experimenty a porovnať výsledky s výsledkami, ktoré boli získané zo štandardného EACircu. Z výsledkov, ktoré boli prezentované v kapitole 4 vyplýva, že EACirc s novým typom uzlov dosahuje podobné výsledky ako EACirc s bežnými uzlami, avšak vo väčšine prípadoch s trochu menšou istotou. To znamená, že zo všetkých behov, ktoré boli spustené bolo viac úspešných práve tých, ktoré používali bežné uzly. Avšak to nemusí znamenať, že EACirc s JVM uzlami nemá žiadnu perspektívu. Dalo by sa nájsť ešte veľa experimentov, v ktorých by mohli JVM uzly prekvapiť.

Ďalšou úlohou bolo overiť, či je pre genetiku výhoda, a teda či je EACirc úspešnejší, ak sa používajú v uzloch inštrukcie z kryptografickej funkcie, ktorej dáta sa práve testujú. Z výsledkov vyplýva, že toto tvrdenie je nepravdivé, pretože napríklad výsledky pre funkciu *Decim* jasne ukazujú, že bytecode z tejto funkcie mal najhoršiu úspešnosť zo všetkých testovaných bytecodov. Dokonca zvládol rozoznať o jednu rundu menej ako všetky ostatné prípady.

Pre niektoré kombinácie som spustil aj výpočty s 300000 generáciami. Tieto výpočty dopadli lepšie ako tie na 30000 generáciách, čo je však očakávané, pretože dlhšie vylepšovanie obvodu by malo logicky vyústiť k nájdeniu úspešnejších obvodov. Veľmi zaujímavé rozšírenie tohto experimentu, čo sa týka počtu generácií, by bolo nájsť taký počet generácií, pri ktorom by sa EACirc už nezlepšoval. To znamená, že ďalšie pridanie generácií by už nezlepšilo úspešnosť a keď áno, tak len minimálne. Takýto počet generácií by sa našiel pre bežné uzly a aj pre JVM uzly. Zaujímavé by to bolo z toho dôvodu, lebo by to mohla byť potencionálna silná stránka JVM uzlov, keďže dokážu vytvoriť zložitejšie obvody ako bežné uzly. Avšak vykonanie takéhoto experimentu by bolo dosť časovo náročné, hlavne pri JVM uzloch, keďže už výpočty pri 30000 generáciách trvajú mnohonásobne dlhšie ako výpočty pre bežné uzly. Takže jediný spôsob ako toto otestovať, je zmeniť implementáciu JVM uzlov tak, aby jeden beh s týmito uzlami bol rýchlejší.

Ďalší ukázaný experiment bol ten, ktorý používal bytecode zo šifrovacej funkcie AES. Výsledky tohto experimentu skončili veľmi podobne ako ostatné výpočty, teda podobnými výsledkami ako dosiahol EACirc s bežnými uzlami, avšak o trochu horšími. Tu sa však ukázali zaujímavé výsledky pre funkciu *Decim*, kde dosiahol EACirc s bytecodom z funkcie AES približne dvakrát lepšie výsledky ako EACirc s ostatnými bytecodmi. Preto by bolo veľmi zaujímavé zistiť, prečo je AES úspešnejší. Toto by sa dalo vykonať hľadaním rozdielu medzi bytecodmi AES-u a ostatných funkcií. Z toho by sa dalo vydedukovať, čo genetike pomáha pri hľadaní lepších obvodov. Či sa jedná o chýbajúcu inštrukciu, ktorú AES obsahuje a ostatné bytecody nie, alebo sa najčastejšie používaná inštrukcia z AES-u vyskytuje v ostatných bytecodoch len veľmi zriedka atď. V konečnom dôsledku by sa dal vytvoriť bytecode len z najčastejšie používanými inštrukciami pre konkrétnu funkciu, ktorý by mohol mať lepšiu úspešnosť. Avšak toto by bolo v rozpore so základnou myšlienkou pre JVM simulátor, pretože predpokladom je, že bytecode sa len jednoducho vytvorí z kryptografickej funkcie, poskytne sa EACircu, a on spraví všetku prácu, čo sa týka vyberania tých správnych inštrukcií. Naneštastie zatiaľ to vyzerá tak, že genetika má problém so skladaním týchto obvodov, aspoň pri bežnom nastavení, aké sme použili.

Tu sa dostávame k otázke, či ponechať bytecode, ktorý sme získali z funkcie v celku, alebo pomôcť genetike a bytecode nejakým spôsobom upraviť. Existujú nástroje, ktoré dokážu hľadať v bytecode zmysluplné časti. V kombinácii so spúšťaním celých funkcií z bytecodu, by sme získali obvody čo vykonávajú zmysluplnejšiu funkcionalitu, avšak taký istý efekt by sme dosiahli, keby sme jednoducho vytvorili nové uzly kde každý by obsahoval tento kus inštrukcií. Zautomatizovanie celého tohto procesu by nebolo jednoduché, pretože by bolo treba začleniť už spomenutý nástroj do frameworku EACirc, tak aby stále platilo to, že EACirc dostane bytecode z kryptografickej funkcie a sám si vyberie inštrukcie, ktoré má v uzloch použiť.

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.

[16] *Baseline experiment*. 2016. URL: http://crcs.cz/wiki/doku.php?id=research:eacirc:baseline-experiments (cit. 2016-05-14).

[Ban98] W. Banzhaf. *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Applications*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann Publishers, 1998. Kap. Genetic Programming as Machine Learning. ISBN: 9781558605107. URL: https://books.google.cz/books?id=1697qefFdtIC.

[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).

[Cen] Centre for Research on Cryptography and Security. Faculty of Informatics Masaryk University. URL: http://www.fi.muni.cz/research/crocs/ (cit. 2016-05-13).

[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).

[FIL18] S. FILIPČÍK. *LaTeX Thesis Style [online]*. Bakalářská práce. 2009 [cit. 2016-05-18]. URL: http://is.muni.cz/th/173173/fi_b/.

[Koz92] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN: 9780262111706. URL: https://books.google.cz/books?id=Bhtxo60BV0EC.

[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).

[SUM13] P. Svenda, M. Ukrop a V. Matyáš. "Towards cryptographic function distinguishers with evolutionary circuits". In: *Security and Cryptography (SECRYPT), 2013 International Conference on*. 2013-07, s. 1–12.

[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.

[Š+12] P. Švenda, M. Ukrop, M. Sýs et al. *EACirc. Framework for autmatic search for problem solving circuit via evolutionary algorithms*. Centre for Research on Cryptography a Security, Masaryk University. 2012. URL: http://github.com/crocs-muni/EACirc (cit. 2016-05-14).

[ŠUM14] P. Švenda, M. Ukrop a V. Matyáš. "E-Business and Telecommunications: International Joint Conference, ICETE 2013, Reykjavik, Iceland, July 29-31, 2013, Revised Selected Papers". In: ed. S. M. Obaidat a J. Filipe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Kap. Determining Cryptographic Distinguishers for eStream and SHA-3 Candidate Functions with Evolutionary Circuits, s. 290–305. ISBN: 978-3-662-44788-8. DOI: 10.1007/978-3-662-44788-8_17. URL: http://dx.doi.org/10.1007/978-3-662-44788-8_17.

[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).

A Príloha

Prílohy sú dostupné v úschovni[1] a obsahujú nasledujúce položky:

- **jvm-simulator**
  Implementácia JVM simulátora
- **konfiguracne-subory**
  Niekoľko príkladov ako vyzerá konfiguračný súbor
- **pouzite-bytecody**
  Všetky bytecody spomenuté v kapitole 4
- **thesis-zdroj**
  Zdrojový kód práce, ktorý obsahuje aj bibliografiu a použité obrázky

1. http://is.muni.cz/th/422190/fi_b/