VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

GENERÁTOR NÁHODNÝCH ČÍSEL

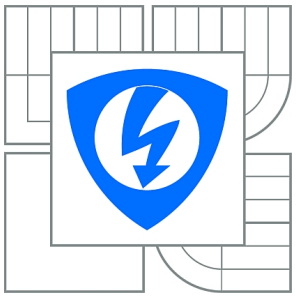
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. VILIAM KRIŽAN
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

GENERÁTOR NÁHODNÝCH ČÍSEL

RANDOM NUMBER GENERATOR

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

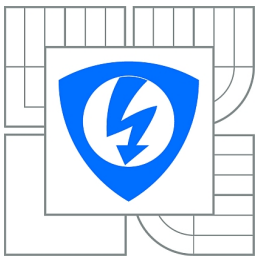
AUTOR PRÁCE — Bc. VILIAM KRIŽAN

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR LEŽÁK

SUPERVISOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. Viliam Križan ***ID:*** 134530
Ročník: 2 ***Akademický rok:*** 2014/2015

NÁZEV TÉMATU:

Generátor náhodných čísel

POKYNY PRO VYPRACOVÁNÍ:

V jazyce Java implementuje kryptograficky silný generátor náhodných čísel Fortuna a začleňte jej do knihovny Aalg. Využijte různé zdroje entropie (systémový generátor náhodných čísel, mikrofon, videokameru, vstup od uživatele, ...).

DOPORUČENÁ LITERATURA:

[1] LEŽÁK, Petr. The abstract algebra library. [online]. [cit. 2014-04-14]. Dostupné z: http://sourceforge.net/projects/aalg/

[2] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Cryptography Engineering: Design Principles and Practical Applications. 2010 [cit.21.10.2014] ISBN: 978-0-470-47424-2

Termín zadání: 9.2.2015 ***Termín odevzdání:*** 26.5.2015

Vedoucí práce: Ing. Petr Ležák
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Táto diplomová práca sa zaoberá generovaním náhodných čísel a implementáciou generátoru Fortuna v jazyku Java. Práca sa najprv venuje teoretickému zoznámeniu s problematikou generácie náhodných dát. Potom sú v nej popísané a analyzované rôzne zdroje entropie ako pohyb myšou, písanie na klávesnici, zachytávanie šumu pomocou mikrofónu a zachytávanie šumu pomocou webkamery. Analýza sa zaoberá náhodnosťou, použiteľnosťou a objemom získaných dát. Potom je v práci spomínaný teoretický opis generátoru náhodných čísel Fortuna. Na záver je predstavený a popísaný objektový návrh programu Fortuna a sú tu spomenuté aj všetky implementačné detaily s tým súvisiace.

KĽÚČOVÉ SLOVÁ

entropia, Fortuna, generátor náhodných čísel

ABSTRACT

This master thesis deals with a generation of random numbers and Fortuna generator implementation in Java language. In the first part the theoretical familiarization to the issues is introduced. Various entropy sources like mouse movement, keyboard typing, microphone and web camera noise are described and analysed. The analysis focuses on randomness, usability and volume of gathered data. Also the Fortuna random number generator is described from the theoretical view. Object analysis and implementation details are described in the last chapter of the document.

KEYWORDS

entropy, Fortuna, random number generator

VILIAM, Križan *Generátor náhodných čísel*: diplomová práca. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2015. 51 s. Vedúci práce bol Ing. Petr Ležák

PREHLÁSENIE

Prehlasujem, že som svoju diplomovoú prácu na tému „Generátor náhodných čísel“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/nebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., o právu autorském, o právoch súvisejúcich s právom autorským a o zmeně niektorých zákonov (autorský zákon), vo znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhé, hlavy VI. diel 4 Trestného zákoníka č. 40/2009 Sb.

Brno ....

(podpis autora)

POĎAKOVANIE

Rád by som sa poďakoval vedúcemu diplomovej práce pánovi Ing. Petrovi Ležákovi za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno ....

(podpis autora)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

POĎAKOVANIE

Výzkum popsaný v tejto diplomovej práci bol realizovaný v laboratóriách podporených projektom SIX; registračné číslo CZ.1.05/2.1.00/03.0072, operačný program Výzkum a vývoj pro inovace.

Brno ....
(podpis autora)

OBSAH

ZOZNAM OBRÁZKOV

ÚVOD

V dnešnej dobe sa venuje ochrane a integrite osobných dát zvýšená pozornosť. Či už chceme vybudovať zabezpečený kanál SSL alebo len zašifrovať dáta na disku, využívame pri tom rokmi overené kryptografické metódy. Avšak tieto metódy potrebujú pre svoju správnu funkčnosť zdroj náhodných dát. Ak preto nevenujeme pri implementácii generátoru náhodných čísel zvýšenú pozornosť, môže sa ľahko stať slabým článkom kryptografického reťazca a vytvoriť tak bezpečnostnú slabinu.

Táto práca sa zaoberá generovaním náhodných čísel. V prvej a druhej kapitole sú vysvetlené základné pojmy, rozdiel medzi generátorom náhodných čísel a generátorom pseudonáhodných čísel, je definovaný pojem entropia a jej vlastnosti. Tretia kapitola sa zaoberá zdrojmi entropie. Ako zdroje entropie boli zvolené myš, klávesnica, webkamera, mikrofón a generátor náhodných čísel v Jave. Jednotlivé zdroje sú analyzované na základe efektívnosti a použiteľnosti. Ďalej sú v tretej kapitole popísané základné princípy generátoru náhodných čísel Fortuna. Štvrtá kapitola sa venuje objektovému návrhu a implementácii generátoru Fortuna. Pri implementácii sa kládol dôraz na funkčnosť, výkonnosť, modularitu a ľahkú rozšíriteľnosť.

1 NÁHODNÉ VS PSEUDONÁHODNÉ ČÍSLA

V tejto kapitole sú popísané tzv. náhodné čísla a pseudonáhodné čísla. Sú v nej popísané kladné aj záporné vlastnosti týchto čísel ako aj ich generácia. Je v nej takisto popísaný spôsob akým sa môžu generátory náhodných a pseudonáhodných čísel kombinovať tak, aby sa navzájom eliminovali ich negatívne vlastnosti.

1.1 Pravé náhodné čísla

Na generáciu zcela náhodných čísel, označovaných aj ako true random numbers, sa využívajú fyzikálne javy. Medzi takéto fyzikálne javy môžeme zaradiť napríklad rozklad jadier rádioaktívneho prvku kedy sa snažíme pomocou Gaigerovho počítača zmerať čas dvoch po sebe nasledujúcich rozpadov jadier. Ďalším príkladom ako získavať náhodné čísla môže byť meranie atmosferického šumu, tepelného šumu apod. Takéto generátory sa v anglickej literatúre nazývajú TRNG(true random number generator)[3][8].

Výhoda takýchto generátorov je generácia zcela náhodných kryptograficky silných čísel. Nevýhoda týchto generátorov je ich závislosť na danom fyzikálnom jave. Generátory často krát nie sú schopné vyprodukovať požadované množstvo náhodných čísel v krátkom čase. Ich závislosť môže využiť aj útočník a generátor môže vystaviť vhodnému typu rušenia, čím môže ovplyvniť výstup generátora v jeho prospech[3][5].

Ďalšou nevýhodou je závislosť na hardvére, ktorú uvediem na príklade generátora náhodných čísel integrovaného na čipoch od firmy Intel [1]. Tento generátor so sebou prináša skrytú implementáciu, tzv. čierna skrinka. Hoci sa na internete dajú dohľadať informácie o funkčnosti tohto generátora, zväčša sú to informácie od firmy Intel a mnoho ľudí sa domnieva, že by tento generátor mohol obsahovať zadné vrátka alebo implementačné chyby. [4][6].

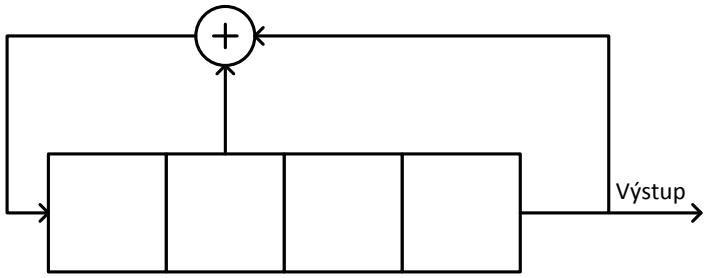
Netreba však zabúdať ani na hardvérovú poruchu generátora. Celková porucha generátora je ešte ten lepší prípad. Môže sa však stať, že sa vyskytne porucha pri ktorej bude generátor funkčný avšak bude produkovať dáta, ktoré nebudú zcela náhodné. Tento typ poruchy sa ťažko odhaľuje a vystavuje nás bezpečnostnému riziku po dobu odhalenia poruchy a následným problémom spojeným s odstraňovaním poruchy[5].

[1] Integrovaný generátor od Intelu je síce pseudonáhodný, avšak využíva len jeden zdroj entropie, ktorý meria tepelný šum na čipe, je teda priamo závislý na jednom fyzikálnom jave.

1.2 Pseudonáhodné čísla

Pseudonáhodné čísla sú čísla, ktoré sú generované deterministicky na základe určitého algoritmu. Výstup generátora je teda závislý na použitom algoritme a na vnútornom stave generátora. Vlastnosti pseudonáhodných generátorov sú také, že sú schopné vyprodukovať obrovské množstvo dát v krátkom čase. Po čase sa však stane, že vnútorný stav generátora nadobudne hodnotu, ktorú mal pri jeho spustení a začne generovať tie isté dáta. Pseudonáhodné generátory teda pracujú v cykloch. Hoci sa môže zdať výstup pseudonáhodného generátora zo štatistického hľadiska ako náhodný. Útočník môže pri znalosti implementácie použitého pseudonáhodného generátora a pri znalosti vnútorného stavu predpovedať jeho výstup [3][5][8].

Príklad môžeme vidieť na obrázku 1.1, na ktorom je znázornený tzv. linear feedback shift register. Jedná sa o veľmi jednoduchý ilustratívny 4 bitový generátor. Pri každom kroku sa hodnota zapísaná v pamäťovej bunke presunie a zapíše do susediacej pamäťovej bunky napravo. Z poslednej pamäťovej bunky získavame výstup, v každom kroku teda jeden bit. Do prvej pamäťovej bunky sa v každom kroku zapíše príslušná hodnota pomocou spätnej väzby. Spätná väzba vedie z druhej a štvrtej bunky a pomocou logickej operácie XOR sa zistí príslušná hodnota bitu, ktorá sa zapíše do prvej pamäťovej bunky[2].



Obr. 1.1: LFSR.

V tabuľke 1.1 môžeme vidieť výstup ak za počiatočný stav zvolíme 1101. Z tabuľky môžeme vyčítať vyššie spomínané vlastnosti. Za prvé, výstup vyzerá naoko náhodne avšak je deterministický. Za druhé, generátor pracuje v cykle. V siedmom kroku nadobudol vnútorný stav generátora hodnotu 0110, ktorá sa už raz vyskytla a generátor teda začína generovať odznova rovnakú postupnosť. Za tretie, generátor nenadobudol počas cyklu všetky možné hodnoty vnútorného stavu, napríklad stav 1111 alebo 1010 sa vôbec nevyskytli. Pritom laicky by sme predpokladali, že pri 4 bitovom generátore bude perióda 16 cyklov, je tomu však omnoho menej[2].

Tab. 1.1: Výstup LFSR.

n	stav	výstup
0	1101	x
1	0110	1
2	0011	0
3	0001	1
4	0000	1
5	1000	0
6	1100	0
7	0110	0

Na prvý pohľad sa môže zdať, že pseudonáhodné generátory majú viac nevýhod ako výhod a že sú na kryptografické účely nepoužiteľné. Ich nevýhody však vieme veľmi ľahko eliminovať. Stačí ak v pravidelných intervaloch výpočet pozastavíme, zmeníme hodnoty vnútorného stavu generátora a znova spustíme výpočet. Týmto spôsobom zamedzíme vzniku cyklov a narušíme deterministickosť výstupu. Z výstupu generátoru teda nebude možné odvodiť jeho budúci výstup alebo jeho predchádzajúci výstup [5].

Akým spôsobom však môžeme meniť vnútorný stav generátoru? Na to môžeme použiť už vyššie spomínané TRNG u ktorého vravíme, že je zdrojom entropie pre pseudonáhodný generátor (PRNG - Pseudo random number generator). Ideálne je použiť hneď niekoľko zdrojov entropie. Necháme niekoľko TRNG generovať dáta. Potom tieto dáta zreťazíme, zhashujeme a výstup n s pevnou dĺžkou použijeme ako ďalší vnútorný stav PRNG. Veľkou výhodou tohto prístupu je redundancia. Prípadná porucha jedného zdroja entropie neovplyvní činnosť generátora. Takisto útočník nie je schopný rušením a ovplyvňovaním jedného zdroja entropie odhadnúť výstup hashovacej funkcie n a tým pádom vnútorný stav generátora [5].

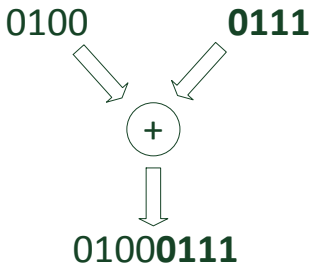
2 ENTROPIA

V kryptografii je entropia chápaná ako miera náhodnosti. 8 bitové plne náhodné číslo má pre útočníka entropiu 8 bitov, správne povedané 8 Shannonov[Sh]. Akákoľvek ďalšia informácia, ktorú má útočník k dispozícii znižuje entropiu čísla. Entropia je matematicky definovaná takto:

$$H(X) := -\sum_{x} P(X = x) \log_2 P(X = x)$$

kde $P(X = x)$ je pravdepodobnosť, že premenná X nadobudne hodnotu x. Dobrou vlastnosťou na entropii je to, že pri kombinovaní neklesá. Túto vlastnosť demonštruje obrázok 2.1. Povedzme, že chceme vytvoriť 8 bitové náhodné číslo a použijeme na to dva generátory. Naľavo sa nachádza zcela náhodné číslo z prvého generátora s entropiou 4 Sh. Vpravo však máme číslo, o ktorom predpokladáme, že je z druhého generátora, avšak toto číslo je podvrhnuté útočníkom a má teda nulovú entropiu. Kombináciou týchto dvoch čísel vznikne 8 bitové číslo s entropiou 4 Sh, pretože pre útočníka toto číslo nadobúda celkovo 2^4 možností. Reálna entropia by bola pravdepodobne ešte o niečo vyššia ako 4 Sh, pretože útočník nemusí vedieť aký spôsobom čísla kombinujeme[5][10].

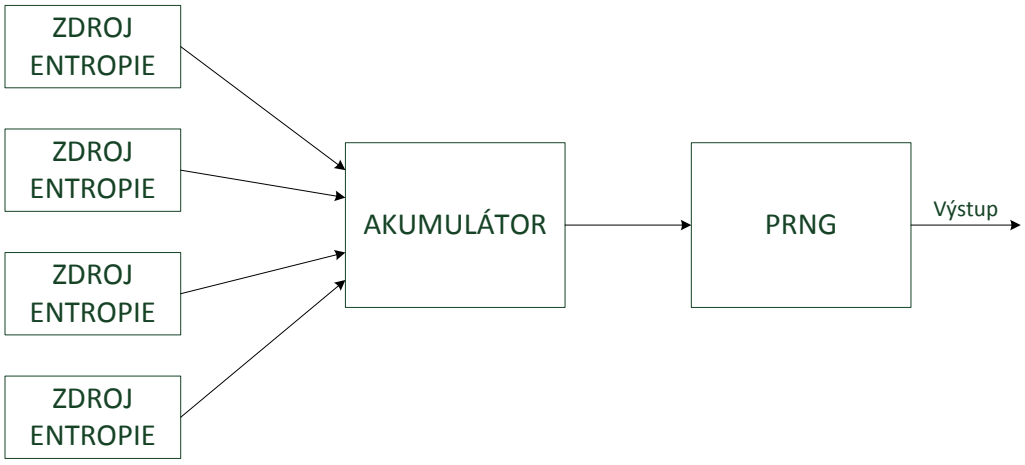
Túto vlastnosť potom samozrejme využívame v praxi. Ak napríklad potrebujeme vytvoriť 16 bitové náhodné číslo, tak potrebujeme nazbierať dáta, ktoré majú entropiu aspoň 16 Sh. Je pritom jedno či tieto dáta pochádzajú zo zarušených zdrojov. Akonáhle nazbierame potrebných 16 Sh entropie v súbore dát povedzme 1000 bitov, tak tento súbor zhešujeme a posledných 16 bitov výstupu hešovacej funkcie nám dá naše 16 bitové náhodné číslo. Reálne však nie sme schopný odhadnúť ktoré zdroje sú rušené útočníkom a do akej miery, nie sme teda schopný určiť entropiu zdroja a preto sa snažíme nazbierať čo najviac dát a predpokladáme pritom, že sme dosiahli hodnotu našej predpokladanej minimálnej entropie.



Obr. 2.1: Kombinácia dvoch zdrojov entropie.

3 FORTUNA

Fortuna je kryptografický generátor náhodných čísel navrhnutý Nielsom Fergusonom a Bruceom Schneierom. Na obrázku 3.1 je znázornené všeobecné blokové schéma kryptografického generátora náhodných čísel z ktorého vychádza aj generátor Fortuna. Generátor sa skladá z troch častí: z pseudonáhodného generátoru, z akumulátoru a zo zdrojov entropie. Zdroje entropie nám slúžia na zber náhodných dát. Tieto dáta sa akumulujú v akumulátore. V pravidelných časových intervaloch sa dáta z akumulátora extrahujú a použijú sa ako nový vnútorný stav pseudonáhodného generátora, ktorý generuje nami požadované náhodné dáta. Jednotlivé bloky sú detailne popísané v samostatných podkapitolách [5].



Obr. 3.1: Schéma generátoru Fortuna.

3.1 Zdroje entropie

Zdroje entropie nám slúžia na zber náhodných dát. Zdroje entropie nie sú presne definované. Za zdroj entropie teda môžeme použiť čokoľvek, čo generuje náhodné dáta, pričom sa snažíme využiť viacero zdrojov naraz a zabezpečiť tak redundanciu systému. Zväčša sa snažíme využiť možnosti, ktoré nám ponúkajú samotné počítače. Zdroje entropie potom môžeme rozdeliť na:

- interaktívne
- neinteraktívne
- systémové

3.1.1 Interaktívne

Interaktívne zdroje entropie využívajú na svoju činnosť užívateľskú interakciu. Medzi takúto interakciu radíme napríklad pohyb s myšou alebo písanie na klávesnici. Veľkou výhodou interaktívnych zdrojov entropie je vysoká náhodnosť spôsobená užívateľom, ktorá je výhodná pri generovaní náhodných čísel. Nevýhodou je napríklad nepravidelnosť s akou sú generované náhodné dáta, ak totiž užívateľ v danom momente nepoužíva ani myš ani klávesnicu tak nie sú generované žiadne dáta. V nasledujúcom texte sú popísané metódy zaoberajúce sa pohybom myši a písaní na klávesnici.

Pohyb myšou

Pohyb myšou je jedna z interaktívnych metód ako generovať náhodné čísla. Dôležité je pritom zvoliť vhodnú metódu ako budú tieto dáta generované. Pravidelné snímanie a zaznamenávanie súradníc kurzoru nie je práve vhodná metóda. Ak totiž užívateľ myš v danom momente nepoužíva tak by sa generovali stále tie isté hodnoty. Vhodné je teda zaznamenávať pohyb a veličiny ako rýchlosť pohybu alebo zmena rýchlosti pohybu teda akceleráciu.

Ja som sa rozhodol pre implementáciu akcelerácie. Na zber dát som vytvoril jednoduchý Javovský program pozostávajúci z troch tried. Trieda MouseMovement je trieda, ktorá obsahuje metódu main. Slúži na spustenie, inak nie je ničím výnimočná. Trieda GUI nám vykreslí okno na celú obrazovku, na ktorom sa bude detekovať pohyb myši. Trieda ďalej obsahuje privátnu triedu HandlerClass ktorá implementuje MouseMotionListener.

```
HandlerClass handler = new HandlerClass();
mousePanel.addMouseMotionListener(handler);
```

MouseMotionListener nám pri každom pohybe myši zavolá metódu mouseMoved, ktorá zavolá metódu addNewCoordinate. Metóda addNewCoordinate je statická metóda triedy MouseData, pri jej zavolaní sa do triedy MouseData uložia aktuálne súradnice kurzoru. V triede MouseData sa nasnímané súradnice ďalej spracovávajú.

```
public void mouseMoved(MouseEvent e) {
   int k = MouseData.addNewCoordinate(e.getX(), e.getY());
```

Matematicky je zrýchlenie definované ako prvá derivácia rýchlosti podľa času. Rýchlosť je zas definovaná ako prvá derivácia polohy podľa času. Zrýchlenie preto môžeme vyjadriť aj ako druhú deriváciu polohy podľa času, viz (3.1).

$$a = \frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\mathrm{d}^2x}{\mathrm{d}^2t} \tag{3.1}$$

Rovnica (3.2) vyjadruje aktuálnu rýchlosť, teda deriváciu polohy podľa času pre spojitý signál.

$$v = \frac{\mathrm{d}x}{\mathrm{d}t} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{3.2}$$

My však pracujeme s diskrétnymi dátami, kde Δt nadobúda určitú hodnotu s ktorou operačný systém zaznamenáva súradnice kurzoru. V takomto prípade sa nám derivácia polohy podľa času zjednoduší na vzťah (3.3).

$$v = \frac{\mathrm{d}x}{\mathrm{d}t} = \frac{x_{t+1} - x_t}{\Delta t} \tag{3.3}$$

Poďme si teraz odvodiť obecný vzťah pre výpočet zrýchlenia. Rozpísaním základného vzťahu, následným dosadením rovnice (3.3) za v_t a upravením získame rovnicu (3.8). Tento vzťah je veľmi dôležitý hlavne z výpočetného hľadiska, pretože sme schopný vypočítať druhú deriváciu v jednom cykle for.

$$a = \frac{\mathrm{d}v}{\mathrm{d}t} \tag{3.4}$$

$$a = \frac{v_{t+1} - v_t}{\Delta t} \tag{3.5}$$

$$a = \frac{\frac{x_{t+2} - x_{t+1}}{\Delta t} - \frac{x_{t+1} - x_t}{\Delta t}}{\Delta t} \tag{3.6}$$

$$a = \frac{(x_{t+2} - x_{t+1}) - (x_{t+1} - x_t)}{(\Delta t)^2} \tag{3.7}$$

$$a = \frac{x_{t+2} - 2x_{t+1} + x_t}{(\Delta t)^2} \tag{3.8}$$

Ak ešte rozšírime našu úvahu o fakt, že Δt je konštantný interval, akým operačný systém zaznamenáva aktuálne súradnice kurzoru myši. Môžeme potom Δt^2 z rovnice (3.8) vyškrtnúť pretože sa jedná o delenie konštantou. Naším primárnym cieľom totiž je generovať náhodné dáta a delením konštantou nezvyšujeme ani neznižujeme náhodnosť výsledných dát. Výsledný vzťah pomocou ktorého analyzujeme nazbierané súradnice kurzoru myši sa nám teda výrazne zjednoduší na rovnicu (3.9).

$$a_n = x_{n+2} - 2x_{n+1} + x_n \tag{3.9}$$

Vzťah (3.9) nám vyjadruje akceleráciu v jednom smere. To nám však nevadí, pretože budeme monitorovať zvlášť akceleráciu v smere osi x a zvlášť akceleráciu v smere osi y.

V tabuľke 3.1 môžeme vidieť namerané hodnoty pri nízkej, strednej a vysokej citlivosti myši. Pri každom meraní bolo zaznamenaných 998 súradníc. Z týchto súradníc sme vypočítali zrýchlenia v smere osi x a y. Vypočítané hodnoty sme spolu zmiešali a následne sme analyzovali výskyt jednotlivých hodnôt zrýchlenia. V tabuľke jasne

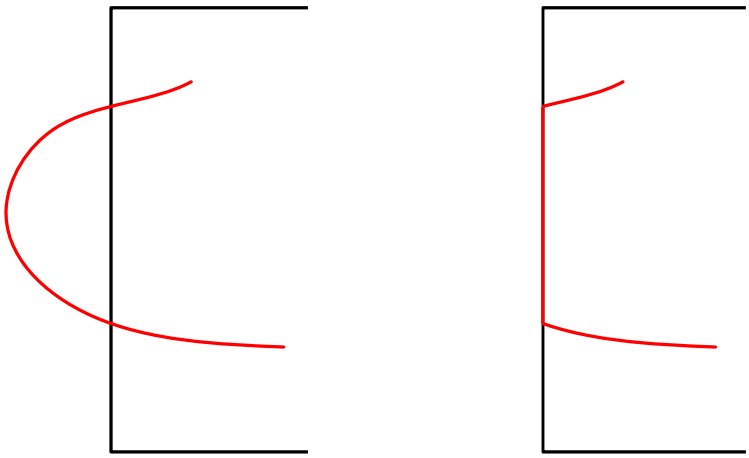
vidieť dominujúce nízke hodnoty, hlavne nulu. Ide o hodnoty pri ktorých kurzor myši vykonával rovnomerný pohyb alebo jemné zmeny rýchlosti. Prudké zmeny rýchlosti nie sú až také časté.

Čo sa týka závislosti na citlivosti tak pri nízkej citlivosti boli nízke hodnoty zrýchlenia najpravdepodobnejšie. Užívateľ by musel v tomto prípade vykonať veľmi prudký pohyb aby docielil vygenerovanie vyššej hodnoty. Logicky by sme mohli predpokladať, že najlepšie výsledky dostaneme pri najvyššej nastavenej citlivosti. Bohužiaľ tomu tak nebolo a myslím si, že dôvodom bol jav konečnej plochy monitora, ktorý zapríčinil zvýšený výskyt nulových sekvencii v meraní. Tento jav bude opísaný v nasledujúcom texte. Najlepšie výsledky sme teda dosiahli pri strednom nastavení citlivosti myši. Toto meranie obsahovalo najmenší percentuálny výskyt hodnoty nula a najvyšší percentuálny výskyt vyšších hodnôt zrýchlenia.

Tab. 3.1: Hodnoty zrýchlenia pri pohybe myšou.

	citlivosť myši					
	nízka		stredná		vysoká	
hodnota vzorku	počet vzorkov	percentuálne zastúpenie	počet vzorkov	percentuálne zastúpenie	počet vzorkov	percentuálne zastúpenie
-5:	2	0,100%	21	1,052%	8	0,401%
-4:	9	0,451%	22	1,102%	7	0,351%
-3:	12	0,601%	34	1,703%	31	1,553%
-2:	65	3,257%	87	4,359%	118	5,912%
-1:	466	23,347%	436	21,844%	306	15,331%
0:	875	43,838%	681	34,118%	753	37,725%
1:	482	24,148%	422	21,142%	385	19,289%
2:	54	2,705%	81	4,058%	169	8,467%
3:	8	0,401%	28	1,403%	61	3,056%
4:	9	0,451%	24	1,202%	22	1,102%
5:	3	0,150%	18	0,902%	9	0,451%
iná:	11	0,551%	142	7,114%	127	6,363%

Monitorovanie pohybu myši so sebou prináša aj ďalšie nevýhody. Na obrázku 3.2 môžeme vidieť rozdiel medzi skutočným pohybom myši a zaznamenanými súradnicami. Naľavo je červenou krivkou znázornený skutočný pohyb akým užívateľ pohol myšou. Napravo je červenou farbou znázornený pohyb kurzoru, ktorý je limitovaný veľkosťou monitora. Ako môžeme vidieť v trajektórii kurzora sa nachádza zvislá čiara. Jednotlivé body zvislej čiary majú rovnakú polohu vzhľadom na os x, preto aj rýchlosť aj zrýchlenie budú vzhľadom na os x nulové. V spracovaných dátach sa



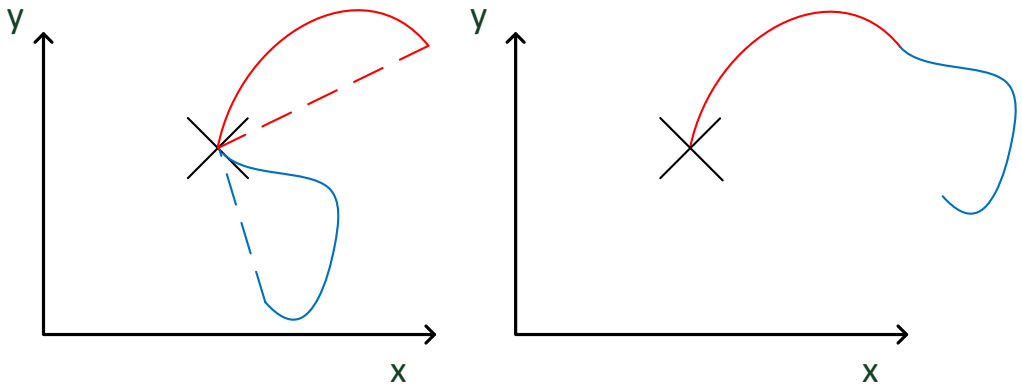
Obr. 3.2: Pohyb myši vs skutočné zaznamenané súradnice.

nám teda objaví sekvencia opakujúcich sa núl tam kde by sa za ideálnych podmienok neobjavila.

Riešenie ako obísť toto obmedzenie môže byť napríklad také, že budeme kurzor myši neustále centrovať na stred. Princíp je podobný ako v počítačových hrách kde máme kurzor respektíve mieritko na strede obrazovky a pri pohybe myšou sa nám mení pozadie.

Princíp ilustruje obrázok 3.3. Krížik nám ilustruje východzí bod. Napravo môžeme vidieť trajektóriu pohybu myši presne tak ako ju vykonal užívateľ. Naľavo je zobrazený spôsob akým pracuje aplikácia, kde počas pohybu myšou došlo k centrovania. Z jednej krivky sme tak dostali dve krivky pričom obe vychádzajú z východziehoho bodu.

Pri každom centrovaní však ukladáme aj offset. Vďaka čomu sme schopný z týchto čiastkových kriviek poskladať tú skutočnú trajektóriu tak ako ju vykonal užívateľ. Princíp akým je toto riešenie implementované je znázornené na obrázku 3.4. Užívateľ hýbe myšou (červená krivka) a dané súradnice kurzoru sa ukladajú. Zrazu sa však vykoná centrovanie. Súradnice koncového bodu červenej krivky b1 sa uložia ako offset. Užívateľ však naďalej hýbe myšou. Tento pohyb je reprezentovaný modrou krivkou. Súradnice modrej krivky sa ukladajú avšak je k nim pripočítaný offset. Do pamäte sa teda ukladajú hodnoty súradníc presne tak aké by boli keby nedochádzalo k centrovaniam. Následne prichádza druhé centrovanie. Súradnice koncového bodu modrej krivky sú pripočítané k offsetu. Aktuálny offset je teda súčtom predchádzajúcich offsetov. Aktuálny offset je reprezentovaný na obrázku bodom o. Keby užívateľ po druhom centrovaním pokračoval v pohybe myšou tak by sa tento aktu-

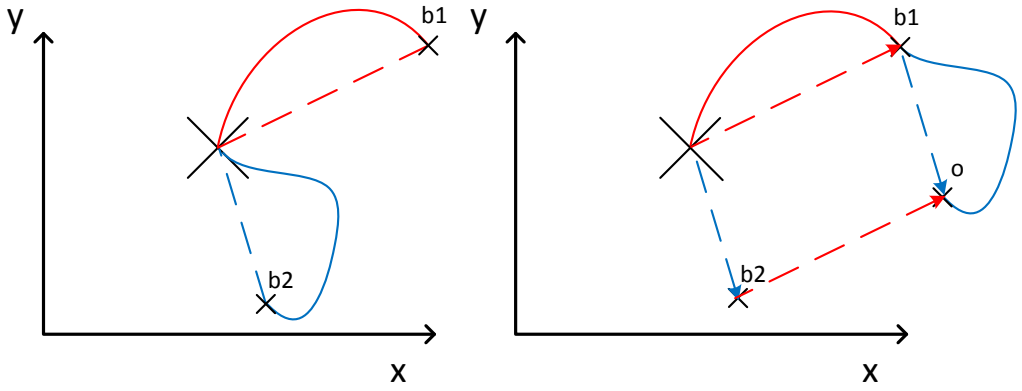


Obr. 3.3: Princíp centrovania.

álny offset pripočítaval k daným súradniciam kurzoru až kým by nedošlo k ďalšiemu centrovaniu.

Tento nový spôsob bol znova otestovali aby sme zistili či došlo k zlepšeniu pri pohybe myšou pri nastavenej vysokej citlivosti. V tabuľke 3.2 môžeme vidieť výsledky. Je zrejmé, že došlo k zlepšeniu a že rozloženie dát vykazuje väčšiu smerodatnú odchyľku.

Poďme sa teraz pozrieť na konkrétnu implementáciu. Na centrovanie myši bola využitá trieda Robot. Trieda Robot nám umožňuje zadávať súradnice a hýbať tak kurzorom myši pomocou programu. Pri tom voláme metódu mouseMove, ktorej zadáme žiadané súradnice x, y kam chceme aby sa kurzor myši pohol. Keďže naša aplikácia sa môže nachádzať kdekoľvek na obrazovke, musíme najprv zistiť polohu



Obr. 3.4: Výpočet aktuálneho offsetu.

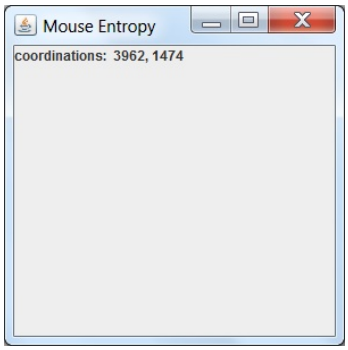
Tab. 3.2: Hodnoty zrýchlenia pri pohybe myšou s použitím triedy Robot.

	citlivosť myši	
	vysoká	
hodnota vzorku	počet vzorkov	percentuálne zastúpenie
-5:	73	1,825%
-4:	91	2,275%
-3:	175	4,375%
-2:	198	4,950%
-1:	200	5,000%
0:	629	15,725%
1:	219	5,475%
2:	177	4,425%
3:	139	3,475%
4:	91	2,275%
5:	66	1,650%
iná:	1942	48,550%

našej aplikácie na obrazovke. Na to využijeme metódu getLocationOnScreen pomocou ktorej sme schopný zistiť súradnice ľavého horného rohu aplikácie. Potom už len stačí získať šírku a výšku okna a pričítať ho k týmto súradniciam, práve na toto miesto budeme kurzor myši centrovať.

Na obrázku 3.5 môžeme vidieť ako aplikácia vyzerá. Bohužiaľ kurzor nie je vidieť pretože sa pri screenshote nezobrazuje. Pri spustení programu program negeneruje žiadne dáta a čaká na to kým užívateľ klikne na hociktorú časť okna aplikácie. Pri tomto kliknutí sa kurzor myši vycentruje na stred okna aplikácie. Užívateľ môže potom hýbať myšou pričom kurzor myši zostáva stále na tom istom mieste a jednotlivé súradnice sa ukladajú poprípade aj zobrazujú na status bare. Ak chce užívateľ túto činnosť prerušiť tak len stáčí kliknút ľavým tlačítko myši a aplikácia prestane zbierať dáta a uvoľní kurzor myši.

Ďalej netreba zabudnúť na to, že aj hýbanie kurzoru pomocou triedy Robot vyvoláva tzv. mouse event. To by však znamenalo, že aj čiarkované čiary z obrázka 3.3 by sa ukladali medzi dáta a offset by vždy skončil v bode centrovania. To však nie je žiadúce. Z implementačného hľadiska bol tento problém ošetrený nastavovaním premennej mouseMoveFlag. V momente kedy užívateľ prvý krát klikne na okno programu tak sa kurzor vycentruje na stred okna a premenná mouseMoveFlag sa nastavený na true. Ak potom užívateľ pohne myšou tak je vyvolaný mouse event



Obr. 3.5: Okno programu MouseES.

a zavolá sa metóda mouseMoved. Na začiatku metódy sa testuje či je mouseMoveFlag nastavený na hodnotu true, ak áno tak sa uložia aktuálne hodnoty súradnice kurzoru. Následne sa nastaví premenná mouseMoveFlag na hodnotu false a vykoná sa centrovanie. Centrovanie vyvolá ďalší mouse event, ale keďže je mouseMoveFlag nastavený na false tak sa v metóde mouseMove nič nevykoná. Potom sa beh programu vracia do metódy mouseMove, ktorú vyvolal hneď prvý mouse event. Na konci tejto metódy sa premenná mouseMoveFlag znova nastaví na hodnotu true.

Čo sa týka objemu získaných dát tak ten je do veľkej miery ovplyvnený užívateľom a tým či užívateľ pohybuje myšou alebo nie. Pri kontinuálnom pohybe myšou a nastavenej strednej citlivosti som však zistil, že x-ová a y-ová súradnica kurzoru je zaznamenaná každých 2,72 ms. To je zaokrúhlene 735 zaznamenaných hodnôt za sekundu.

Hoci je pohyb myši spôsobený užívateľom plne náhodný, tak namerané dáta nevykazujú uniformné rozloženie. Je zrejmé, že nízke hodnoty zrýchlenia majú väčšiu pravdepodobnosť výskytu ako hodnoty vyššie čo je tak trocha slabinou tejto metódy. Pravdepodobnostné rozloženi ďalej ovplyvňujú faktory ako citlivosť myši a užívateľ samotný. Za výhodu tejto metódy považujem jej dostupnosť. Takmer každý užívateľ používa pri práci s osobným počítačou myš alebo touchpad. Ako bude popísané v ďalšou texte tejto práce, tak zariadenia ako mikrofón a webkamera sa dajú lepšie využiť na tvorbu náhodných dát avšak nie každý stolný počítač disponuje webkamerou alebo mikrofónom.

Stisk klávesy

Ďalším interaktívny spôsobom ako získavať náhodné dáta od užívateľa je monitorovanie kláves. Pri stisknutí klávesy môžeme zaznamenať to aká klávesa bola stisknutá a čas kedy bola stisknutá. Bohužiaľ v drvivej väčšine prípadov užívateľ využíva len určitú časť kláves zatiaľ čo niektoré nevyužíva vôbec. Výhodné je preto zaznamenávať čas stisknutia klávesy. Nie však čas vo formáte hodina:minúta:sekunda, ale len

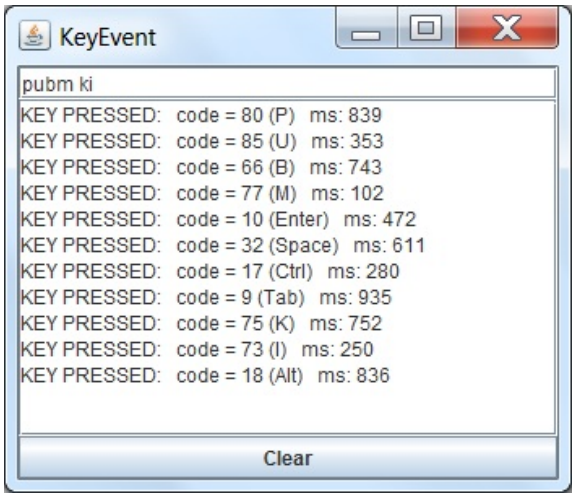
milisekundy po uplynutí sekundy[5].

Na analýzu tejto metódy som vytvoril program, ktorého výstup je možné vidieť aj na obrázku 3.6. Program implementuje rozhranie KeyListener. Pri každom stisku klávesy je zavolaná metóda KeyPressed, ktorá zaznamená a zobrazí kód stisknutej klávesy a čas v milisekundách.

Čas v milisekundách sa zaznamenáva takým spôsobom, že sa vždy vytvorí objekt date, ktorý zaznamená čas. Cez objekt date potom zavoláme metódu getTime, ktorá nám vráti počet milisekúnd, ktoré uplynuli od 1. januára 1970. Pomocou operátoru modulo vyfiltrujeme výstup na posledné 3 dekadické čísla, ktoré následne uložíme do premennej milis.

```
date = new Date();
int milis = (int) (date.getTime() % 1000l);
```

Na obrázku 3.6 teda môžeme vidieť, že každým stisknutím klávesy získame náhodné číslo v rozshasu 0-999.

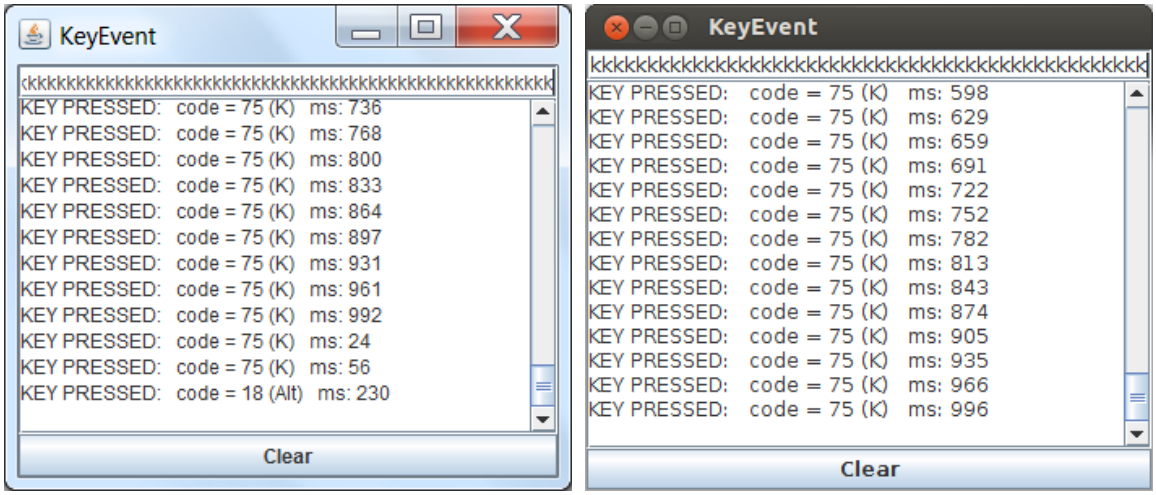


Obr. 3.6: Čas zaznamenaný pri stisku klávesy.

Dôležité je však aj zistiť, či nedochádza ku kešovaniu kláves. Zjednodušene povedané, kešovanie kláves je jav, pri ktorom sa stlačené klávesy ukladajú do medzipamäte. Operačný systém potom načíta tieto dáta z pamäte a priradí im rovnakú časovú značku. Operačný systém teda vníma naše písanie spôsobom ako keby sme stláčali viacero kláves naraz a nie jednu po druhej. Toto môže byť problém pretože sa nám vyskytnú úseky opakujúcich sa čísel a dáta strácajú na náhodnosti.

Tento jav som sa rozhodol experimentálne overiť na mojom počítači s procesorom Intel i3 330M a to tak, že som zámerne držal jednu klávesu po určitú dobu. Na obrázkoch 3.7a a 3.7b môžeme vidieť, že pod operačnými systémami Windows 7 a Ubuntu 12.04 nedochádza ku kešovaniu, pretože každý znak má priradenú špecifickú

časovú značku. Ku kešovaniu nedochádzalo dokonca ani v prípade keď som zámerne simuloval zaťaženie procesora. Záťaž bola simulovaná ďalšími siedmimi bežiacimi vláknami v aplikácii plus cielene bolo spúšťaných veľké množstvo iných aplikácií.



(a) Windows 7 (b) Ubuntu 12.04

Obr. 3.7: Kešovanie kláves.

Čo sa týka náhodnosti dát, netreba zabúdať na to, že užívateľ môže negatívne ovplyvňovať výstup. Ak totiž útočník disponuje znalosťou, že užívateľ vie písať na klávesnici rýchlo, môže toho využiť. Ak napríklad užívateľ stlačí klávesu ktorej je priradená časová značka 400 ms je veľmi pravdepodobné, že pri rýchlosti jeho písania bude mať nasledujúca stlačená klávesa priradenú časovú značku v rozsahu 400-800 ms a naopak časová značka 111 ms je veľmi nepravdepodobná. Dáta potom strácajú mierne na náhodnosti.

Riešením by potom mohlo byť, že by sme časovú značku vyjadrenú v binánej podobe orezali z 10 bitov na povedzme 8 bitov. Časová značka by potom nadobúdala hodnoty 0-255. Jedná sa o kompromis pri ktorom by sme do určitej miery eliminovali vyššie spomínaný jav a zároveň by sme zabezpečili, že sa bude generovať dostatočné množstvo dát pri stlačení klávesy.

3.1.2 Neinteraktívne

Neinteraktívne zdroje entropie sú také, ktoré nevyžadujú užívateľskú interakciu. Ak máme napríklad laptop so vstavanou webkamerou a mikrofónom, môžeme ich využiť na generáciu náhodných dát. V nasledujúcom texte je popísaný spôsob ako získavame dáta z webkamery a mikrofónu a ako z týchto dát získavame šum, ktorý následne analyzujeme z hľadiska náhodnosti.

Šum z mikrofónu

Zachytávanie šumu pomocou mikrofónu je veľmi výhodné, pretože sme schopný získať veľké množstvo náhodných dát v krátkom čase. Samotná Java disponuje API pomocou ktorého sme schopný zachytávať zvuk cez mikrofón. Pre zachytávanie zvuku som sa rozhodol použiť nasledujúce parametre: Vzorkovací kmitočet bol zvolený na hodnotu 192 kHz. Jedná sa o nami najvyšší možný podporovaný vzorkovací kmitočet. Všeobecne by sme mohli povedať, že čím vyšší vzorkovací kmitočet máme, tým viac dát získame. Z týchto dát potom extrahujeme šum. Pri vzorkovaní sa používa metóda PCM a každá vzorka je vyjadrená 16-timi bitmi.

Na zachytávanie audio dát bol vytvorený program CaptureMicrophoneData. Nižšie zobrazený zdrojový kód ukazuje metódu captureAudio. V tejto metóde sa najprv nastaví objekt audioFormat triedy AudioFormat zavolaním metódy getAudioFormat. Konkrétne sa nastaví vzorkovací kmitočet, mono kanál, veľkosť vzorky a formát PCM. Ďalej sa nastaví objekt targetDataLine, pre bližšie informácie viz Java Documentation [7] a [1]. Nakoniec sa vytvorí a spustí vlákno captureThread.

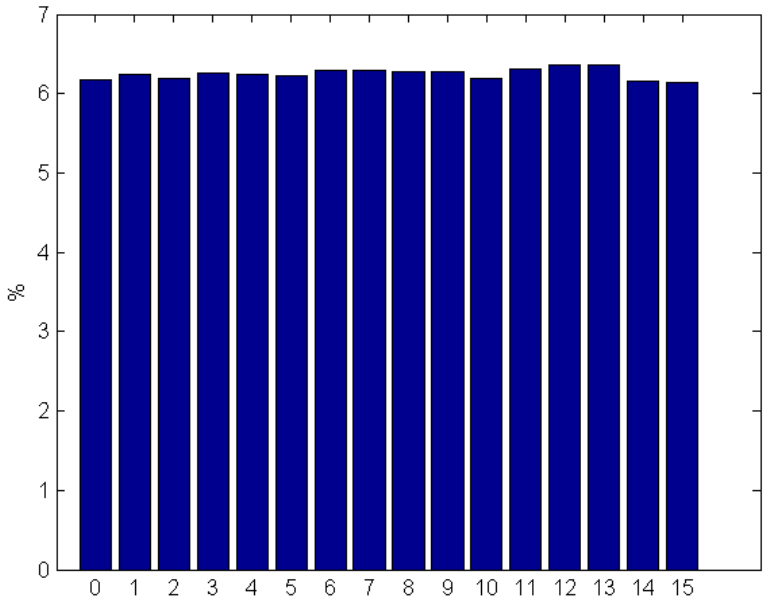
```
private void captureAudio() {
   try {
      audioFormat = getAudioFormat();
      DataLine.Info dataLineInfo = new
         DataLine.Info(TargetDataLine.class, audioFormat);
        targetDataLine = (TargetDataLine)
            AudioSystem.getLine(dataLineInfo);
        targetDataLine.open(audioFormat);
        targetDataLine.start();

        Thread captureThread = new Thread(new CaptureThread());
        captureThread.start();
        } catch (Exception e) {
            System.exit(0);
        }
}
```

Vo vlákne captureThread následne prebieha samotné zachytávanie dát vo vnútri nekonečnej slučky while, ktoré je prerušená až stisknutím tlačítka stop button. Dáta sa v pravidelných intervaloch ukladajú do bufferu pomocou objektu targetDataLine. Z bufferu sú zachytené dáta ukladané priamo do textového súboru volaním metódy saveDataToText, kde ich môžeme bližšie analyzovať. Metóda saveDataToText ešte pred uložením oreže dáta len na posledné najmenej významné bity, ktoré uloží.

```
while (!stopCapture) {
   int cnt = targetDataLine.read(tempBuffer,0,tempBuffer.length);
   if (cnt > 0) {
      saveDataToText(tempBuffer, cnt);
    }
}
```

Na analýzu šumu sme najmenej významné bity zoskupily do štvoríc aby sme zistily či zachytené dáta neobsahujú za sebou dlhé sekvencie núl a jedničiek. Následne sme štatisticky analyzovali tieto štvorice. Na obrázku 3.8 môžeme vidieť histogram znázorňujúci percentuálne zastúpenie jednotlivých štvoríc v meraní v ktorom sme zachytili 425984 najmenej významných bitov. Ako môžeme vidieť, v zachytených dátach sa rovnomerne vyskytujú všetky hodnoty. Jedná sa teda o skutočný šum s vysokou mierou náhodnosti.



Obr. 3.8: Histogram šumu.

Šum získaný pomocou mikrofónu preto hodnotím ako veľmi kvalitný zdroj náhodných dát. Veľkou výhodou je schopnosť generovať veľký objem dát pričom veľkosť dát je závislá od podporovaného kmitočtu. Objem získaných dát môžeme napríklad aj zväčšiť takým spôsobom, že nebudeme využívať len posledný najmenej významný bit, ale posledné dva najmenej významné bity. Pri vzorkovaní PCM so 16-timi bitmi na vzorku by mali mať zachytené dáta stále charakter šumu. Ďalšie pridávanie bitov však bude mať čím ďalej tým väčšie negatívny vplyv na náhodnosť dát.

Šum z webkamery

Webkamera je po mikrofóne ďalší nástroj pomocou ktorého sme schopný získavať neinteraktívne náhodné dáta. Zameriavame sa pritom opäť na šum. Java nám však neposkytuje vstavané knižnice pomocou ktorých by sme boli schopný zachytávať dáta cez webkameru. Možností ako pracovať s webkamerou je viacero, najznámejšia je asi JavaCV, ktorá však vyžaduje inštaláciu openCV. Ja som sa nakoniec rozhodol využiť knižnice projektu sarxos [9], ktoré nevyžadujú žiadne ďalšie doplňujúce inštalácie a poskytujú jednoduché a intuitívne API.

Na získavanie a analyzovanie šumu bol vytvorený program CaptureWebcamData. V programe captureWebcamData bol analyzovaný šum webkamery ako aj štatistické vlastnosti šumu nasnímaných obrázkov. Na analýzu šumu webkamery bol použitý spôsob pri ktorom sa uložia dva snímky statickej scény s odstupom 20 ms. Odstup 20 ms som zvolil z nasledujúcich dôvodov. Za prvé som chcel aby boli obrázky nasnímané v čo najkratšej dobe za sebou, aby boli čo najviac identické a aby čo i len nepatrná zmena osvetlenia nespôsobila zmenu hodnôt pixelov druhého obrázka. Na druhej strane som však nechcel voliť extrémne nízku hodnotu z dôvodu regenerácie snímača CCD.

Na obrázkoch 3.9a a 3.9b môžeme vidieť spomínané obrázky zachytené webkamerou. Ako môžeme vidieť, obrázky sú na pohľad voľným okom totožné. Avšak porovnávaním jednotlivých pixeloch môžeme zistiť, že zo 76800 pixelov až 74498 pixelov obrázka 3.9b nadobúda inú hodnotu ako pixely obrázka 3.9a čo činí až 97,00 %.

(a) Obrázok 1 (b) Obrázok 2

Obr. 3.9: Analyzované obrázky.

Z tohto môžeme vyvodiť záver, že šum webkamery je naozaj veľký a prejavuje sa aj pri snímaní statickej scény, kedy by mali byť nasnímané obrázky teoreticky totožné. Treba si však aj uvedomiť, že meranie som uskutočnil len na jednej webkamere

a iná kvalitnejšia webkamera by preto mohla generovať menší šum. Avšak v reálnej situácii nebude webkamera snímať statickú scénu, ale s vysokou pravdepodobnosťou dynamickú scénu v čase keď užívateľ pracuje za počítačom. Preto je veľmi nepravdepodobné, že sa budú snímať obrázky, ktoré budú mať rovnáke hodnoty všetkých pixelov.

Tabuľka 3.3 nám ukazuje šum extrahovaný z obrázkov 3.9a a 3.9a. Každý pixel obrázka je vyjadrený 24 bitovou hodnotou, ktorá reprezentuje farbu vo farebnom modely RGB (Red Green Blue). Každá zložka, teda červená, zelená a modrá je vyjadrená jedným bytom. Z tohoto bytu extrahujeme posledný najmenej významný bit. Z každého pixelu sme teda schopný extrahovať tzv. tri šumové bity, ktoré môžu nadobúdať hodnotu 0-7. Tabuľka 3.3 nám teda ukazuje počet a percentuálne zastúpenie jednotlivých šumových tribitov. Ako môžeme vidieť získané dáta majú charakter šumu a sú rovnomerne zastúpené.

Tab. 3.3: Analýza šumu obrázkov.

	Obrázok a		Obrázok b	
hodnota vzorku	počet vzorkov	percentuálne zastúpenie	počet vzorkov	percentuálne zastúpenie
0	8913	11,605%	9050	11,784%
1	9769	12,720%	9618	12,523%
2	9119	11,874%	8828	11,495%
3	9734	12,674%	9722	12,659%
4	10367	13,499%	10737	13,980%
5	8916	11,609%	8867	11,546%
6	10513	13,689%	10427	13,577%
7	9469	12,329%	9551	12,436%

Poďme sa teraz bližšie pozrieť na implementáciu v programovacom jazyku Java. Nižšie zobrazený kód nám ukazuje konštruktor, v ktorom sa hneď po spustení programu detekuje a spustí webkamera. Ak žiadna webkamera nie je detekovaná tak sa program ukončí.

```
public CaptureWebcam(){
   try {
      webcam = Webcam.getDefault();
   } catch (Exception e) {
      System.exit(0);
   }
   webcam.open();
}
```

Následne sa zachytia dva snímky s odstupom 20 ms kde objekty image1 a image2 sú objektami triedy BufferedImage. Východzie nastavené rozlíšenie snímaných obrázkov je 240×320.

```
image1 = webcam.getImage();
   try {
      Thread.sleep(20);
   } catch (InterruptedException ex) {
}
image2 = webcam.getImage();
```

Jednotlivé hodnoty pixelov sa získavajú v zanorenej slučke for, kde sa pixely načítavajú riadok po riadku, stĺpec po stĺpci. Hodnota každého pixelu je vyjadrená 32 bitovou integerovou hodnotou. Prvých 8 bitov tvorí tzv. alfa kanál, ďalších 24 bitov reprezentuje RGB hodnotu pixelu o ktorej sme sa už zmieňovali. Keďže objekt triedy BufferedImage neobsahuje metódu, ktorá by nám dovolovala získať zvlášť hodnoty červenej, zelenej a modrej zložky pixelu. Vytvoríme si najprv objekty c1, c2 triedy Color ktoré nám budú reprezentovať pixel. Objekt c1 reprezentuje pixel obrázka 3.9a, c2 zase obrázka 3.9b. Pri vytváraní týchto objektov vložíme do konštuktora 32 bitovú hodnotu, ktorú nám vráti metóda getRGB zavolaná z objektu image1,2 pre daný riadok a stĺpec. Následne z objektov c1, c2 získame jednotlivé zložky farebného modelu RGB volaním metód getRed, getGreen, getBlue, ktoré priradíme do premenných r, g, b a uložíme volaním metódy saveDataToText, krorá uloží posledný bit premenných r, g, b. Na záver ešte oba pixely porovnáme metódou equals aby sme zistili podobnosť obrázkov, ak sú pixely totožné tak je inkrementovaný čítač dpc.

```
int r,g,b;

for (int i = 0; i < image1.getHeight(); i++) {
   for (int j = 0; j < image1.getWidth(); j++) {
      Color c1,c2;
      c1 = new Color(image1.getRGB(j, i));
      c2 = new Color(image2.getRGB(j, i));

      r = c1.getRed();
      g = c1.getGreen();
      b = c1.getBlue();
      saveToText(r, g, b);

      r = c2.getRed();
      g = c2.getGreen();
      b = c2.getBlue();
      saveToText2(r, g, b);

      if(!c1.equals(c2)){
         dpc++;
      }
}
```

Šum získaný pomocou webkamery hodnotím ako veľmi dobrý zdroj náhodných dát. Pri rozlíšení 240×320 sme schopný získať 28,8 kB náhodných dát z jedného zachyteného obrázka. Aplikácia bola schopná na mojom počítači s procesorom Intel i3 330M spracovať jeden zachytený obrázok priemerne za 130 ms. Touto metódou sme teda schopný získať 221,54 kB náhodných dát za sekundu. Limitujúcim faktorom je v tomto smere teda výpočetný výkon. Pri vyššom výpočetnom výkone by sme boli schopný spracovať viac obrázkov a tým pádom získať väčší objem dát.

3.1.3 Systémové

Medzi systémové zdroje entropie by sme mohli zaradiť diskové operácie, špeciálne prerušenie, v niektorých prípadoch aj zaznamenávanie sekvenčných čísel pri naviazaní TCP spojení. Všetky tieto udalosti sú zaznamenávané operačným systémom. K týmto dátam v rámci jazyka Java nemáme ako pristupovať, možeme však využiť implementovaný kryptografický generátor náhodných čísel SecureRandom, ktorý využíva túto systémovú entropiu. V zdrojovom kóde môžeme vidieť príklad v ktorom vygenerujeme 20 bytov náhodných dát pomocou generátora SecureRandom[5][10].

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[20];
bytes = random.generateSeed(20);
```

3.2 Akumulátor

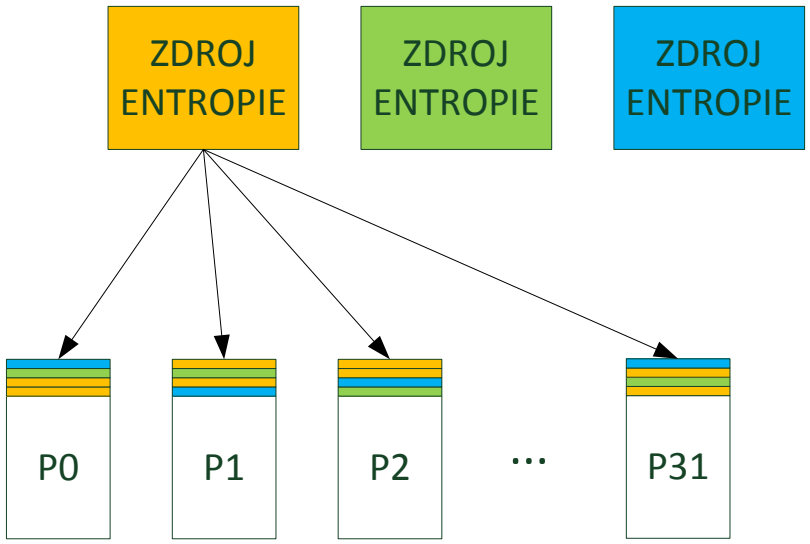
Jednotlivé zdroje entropie zbierajú náhodné dáta, ktoré ukladajú do akumulátora. Dáta z akumulátora sú pravidelne extrahované a použité ako nový vnútorný stav pseudonáhodného generátora[5].

V generátore Fortuna je akumulátor tvorený 32 poolmi. Každý zdroj entropie zbiera entropiu a cyklicky ju prerozdeluje medzi všetky pooly. Dáta sa do poolu ukladajú sekvenčne za sebou, pričom jeho veľkosť je neobmedzená. Pri extrakcii dát z poolu sú dáta z poolu vymazané[5].

Vnútorný stav generátoru meníme vždy keď sa v poole P_0 nahromadí dostatok entropie. Pri každej zmene vnútorného stavu generátora inkrementujeme čítač r. Hodnota čítača r nám zároveň aj udáva, ktoré pooly budú použité pri extrakcii dát z akumulátora. Pool P_i je použitý v prípade ak r je deliteľné 2^i. Pool P_0 je teda použitý vždy, pool P_1 je použitý v každom druhmo prípade, pool P_2 v každom štvrtom prípade atď[5].

Za dostatok entropie nahromadenej v poole P_0 môžeme považovať 128 Sh. Naše zdroje entropie však nemusia generovať čiste náhodné dáta alebo môžu byť rušené útočníkom, preto je veľmi ťažké odhadnúť kolko bitov sa musí v poole P_0 nazbierať aby sme mohli z akumulátora znova extrahovať dáta. V praxy môžeme za predpokladu, že najviac každý druhý bit bude zarušený stanoviť túto hodnotu na 256 bitov. Predpokladajme teraz situáciu, že útočník nejakým spôsobom zistil vnútorný stav generátora a že ruší alebo zámerne generuje veľké množstvo náhodných udalostí v snahe ovplyvniť budúci výstup generátora. Potom existuje pravdepodobnosť, že sa v poole P_0 nestihne nazbierať dostatok entropie a výstup generátora nebude dostatočne kryptograficky bezpečný. Avšak samotná architektúra akumulátora je navrhnutá tak, aby sa generátor bol schopný zregenerovať po určitom čase z narušeného stavu do plne bezpečného stavu. Zatiaľ čo pool P_0 je použitý pri každej extrakcii tak pool P_1 je použitý len pri každej druhej a taký pool P_2 len pri každej štvrtej atď. Preto ak sa za čas t nestihlo nazbierať v poole P_0 dostatok entropie je dosť pravdepodobné, že pool P_1 za čas $2t$ už nazbieral dostatok entropie a keby aj nie tak pool P_2 za čas $4t$ nazbieral ešte viac entropie. Ak preto máme aspoň jeden validný zdroj entropie tak generátor je schopný sa zregenerovať do bezpečného stavu[5].

Posledným bezpečnostným opatrením je, že dáta sa nesmú z akumulátora extrahovať v intervaloch kratších ako 100 ms. Útočník totiž môže generovať veľa náhodných udalostí, z tohto pohľadu sa môže zdať, že sa v pooloch nahromadilo dostatok dát a z akumulátora je možné extrahovať dáta, nie je tomu však tak pretože tieto dáta nie sú náhodné. Ak však zavedieme podmienku, že dáta sa môžu z akumulátoru extrahovať až po uplinutí 100 ms od poslednej extrakcie samozrejme za predpokladu prvotnej podmienky, že v akumulátore je dostatok dát, tak je potom veľmi pravdepodobné, že sa v akumulátore za tých 100 ms popri podstrčených dátach stihlo nazbierať dostatok entropie od nezarušených zdrojov. Pri tejto podmienke sa môžu dáta z akumulátoru extrahovať maximálne 10× za sekundu čo znamená, že pool P_{31} sa použije každých necelých 14 rokov. Z tohoto dôvodu preto nemá zmyseľ rozširovať 32 poolovú architektúru na viac poolov v snahe zvýšenia bezpečnosti[5].



Obr. 3.10: Schéma akumulátoru.

3.3 Pseudonáhodný generátor

Ako pseudonáhodný generátor sa používa bloková šifra. Pritom je jedno aká blokova šifra sa použije. V Našom prípade bude použitá bloková šifra AES (Advanced Encryption System) v režime čítača (CTR režim)[5].

Na obrázku 3.11 je znázornená všeobecná schéma funkčnosti blokovej šifry v režime CTR. Šifrovaná správa sa rozdelí na bloky Z_1 až Z_n. Blok X_n vyjadruje hodnotu čítača. Táto hodnota čítača sa zašifruje v blokovej šifre. Na zašifrovaný výstup (128

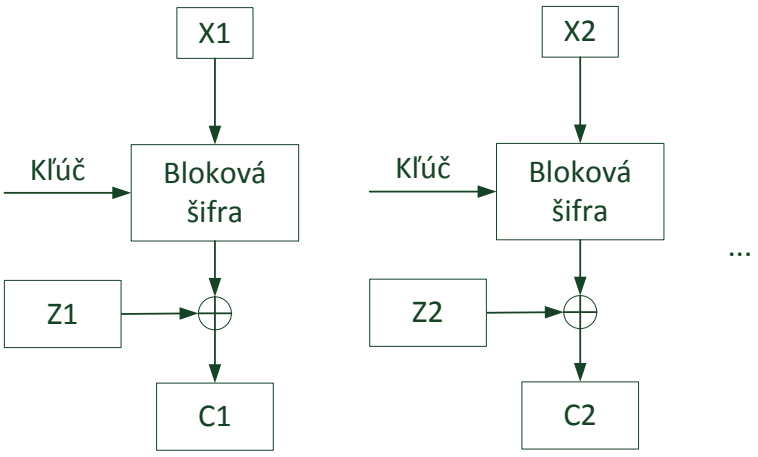
bitov) a blok správy je potom aplikovaná logická operácia XOR, ktorá nám dáva výsledný kryptogram C. V našom prípade však nebudeme šifrovať žiadne správy, preto výstup blokovej šifry je aj celkovým výstupom[5].

Hodnota kľúča bude reprezentovaná 256 bitmi a hodnota čítača bude reprezentovaná 128 bitmi. Po každej žiadosti o generáciu náhodných dát, bude vygenerovaných ďalších extra 256 bitov, ktoré použijeme ako nový kľúč. Ak by sa aj útočníkovy podarilo kompromitovať generátor, nebude schopný zistiť predchádajúci stav generátora[5]. Berúc do úvahy kolíznosť blokov a štatistické vlastnosti vygenerovaných dát autori generátoru Fortuna Bruce Schneier a Niels Ferguson ďalej doporučujú aby na jeden klúč nemalo byť vygenerovaných viac ako 2^{20} bytov dát, čo je presne 1 MiB. Ďalším doporučením je, že pri zmene kľúča nikdy neresetujeme čítač[5].

Nový kľúč K sa vypočíta pomocou hashovacej funkcie SHA-256. Na vstup hashovacej funkcie privedieme starý kľúč K zreťazený s náhodnými dátami s, ktoré sme extrahovali z akumulátora[5].

$$K \leftarrow \mathrm{SHA}_d\text{-}256(K \,||\, s) \tag{3.10}$$

Objem náhodných dát extrahovaných z akumulátora má premenlivý charakter, jednak záleží na tom ako dlho zdroje entropie zbierali entropiu a ešte na hodnote čítača, ktorý určuje z koľkých poolov bude entropia extrahovaná. Ako už bolo spomenuté pri teoretickom opise akumulátora, jednotlivé pooly majú neobmedzenú veľkosť. Z praktického aj bezpečnostného hľadiska je preto nežiadúce aby sme nazbieranú entropiu ukladali do operačnej pamäte alebo dokonca na pevný disk. Riešením je preto výpočet hashovacej funkcie inkrementálnym spôsobom[5].



Obr. 3.11: Bloková šifra v režime CTR.

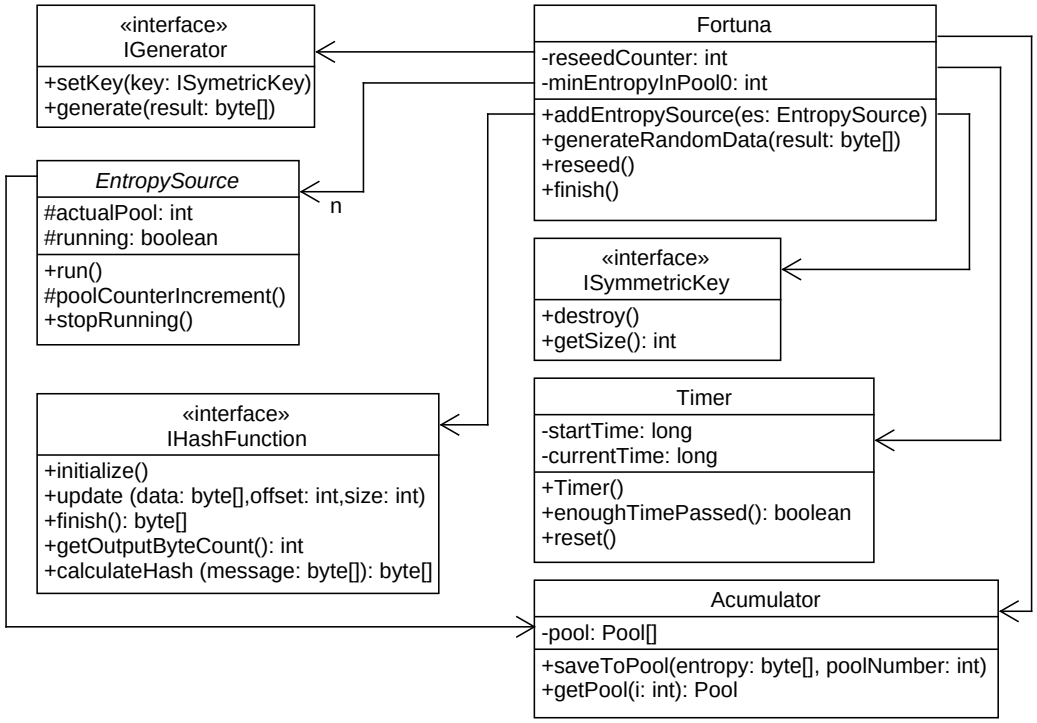
4 IMPLEMENTÁCIA

Táto kapitola sa venuje konkrétnej implementácii generátoru Fortuna v programovacom jazyku Java. Implementácia bude predstavená pomocou diagramov tried v jazyku UML(Unified Modeling Language). Potom bude popísaná funkcionalita jednotlivých tried. Pri implementácii boli použité poznatky objektového návrhu za účelom čo najväčšej nezávislosti medzi komponentami navzájom a za účelom čo najväčšej škálovatelnosti a rozšíriteľnosti.

Pri implementácii sa kládol veľký dôraz na modularitu. Generátor Fortuna je preto schopný pracovať s rôznymi typmi pseudonáhodného generátoru. Takisto môžeme použiť rôzne zdroje entropie a pri výpočte hešu z entropie môže byť použitá ktorákoľvek hešovacia funkcia.

4.1 Systém Fortuna

Na obrázku 4.1 môžeme vidieť diagram tried pre aplikáciu generátora Fortuna pričom sa jedná o tzv. big picture. Je na ňom znázornená hlavná trieda Fortuna obsahujúca ďalšie triedy a moduly (rozhrania).



Obr. 4.1: Diagram tried generátoru Fortuna.

Trieda Fortuna

Trieda Fortuna je hlavná trieda s ktorou pracujeme keď chceme generovať náhodné dáta. Táto trieda obsahuje celkovo osem atribútov. Sú to generator rozhrania IGenerator, hash rozhrania IHashFunction, zoznam zdrojov entropie vo forme array listu, acumulator triedy Acumulator, timer triedy Timer a dve integerové premenné reseedCounter a minEntropyInPool0.

Pri vytváraní objektu z triedy Fortuna voláme konštruktor, ktorému do parametru predáme pseudonáhodný generátor, s ktorým bude fortuna pracovať.

```
BlockCipherGenerator aesCipherGenerator = new BlockCipherGenerator(new
    AesCipher(32));
Fortuna fortuna = new Fortuna(aesCipherGenerator);
```

Fortuna teda môže pracovať s rôznymi pseudonáhodnými generátormi. V konštruktore sa takisto inicializuje pole keyBytes na samé nuly a pomocou tohto poľa potom vytvoríme kľúč z triedy ByteArraySymmetricKey.

```
key = new ByteArraySymmetricKey(keyBytes);
```

Pomocou tohto kľúča nastavíme vnútorný stav generátoru. Ďalej sa v konštruktore inicializuje timer, ktorý slúži na meranie času medzi dvoma reseedmi. Pod pojmom reseed myslíme zmenu vnútorného stavu pseudonáhodného generátora. Ešte netreba zabudnúť na to, že v konštruktore sa inicializuje hešovacia funkcia pomocou ktorej budeme hešovať nazbieranú entropiu. Na to využijeme hešovaciu funkciu SHA512 z knihovny aalg.

Trieda Fortuna však obsahuje aj ďalší konštruktor, ktorý nám umožňuje predať do Fortuny takú hešovaciu funkciu s akou si prajeme pracovať. Tento konštruktor v sebe volá prvý zmieňovaný konštruktor čím sa zamedzí duplicite kódu a predá atribútu hash danú hešovaciu funkciu.

```
public Fortuna(IGenerator generator, IHashFunction hash){
   this(generator);
   this.hash = hash;
}
```

Trieda Fortuna ďalej obsahuje štyri metódy. Prvou Metódou je metóda addEntropySource. Do parametru tejto metódy vložíme zdroj entropie ktorý chceme generátoru Fortuna predať. Vo vnútri metódy najprv pridáme zdroj entropie do zoznamu a potom mu predáme akumulátor, aby mal kam ukladať nazbieranú entropiu. Keďže každý zdroj entropie implementuje rozhranie Runnable tak z neho najskôr vytvoríme vlákno a nastavíme toto vlákno tak aby bežalo na pozadí kde bude zbierať

entropiu. Nakoniec toto vlákno spustíme.

```
public void addEntropySource(EntropySource src){
   etropySource.add(src);
   src.passAcumulatorToES(acumulator);
   Thread th = new Thread(src, "Entropy thread for " +
       src.getClass().getName());
   th.setDaemon(true);
   th.start();
}
```

Druhou metódou je metóda generateRandomData. Do parametru tejto metódy predávame pole bytov do ktorých chceme zapísať náhodné dáta. Táto metóda je vskutku jednoduchá a jej úlohou je to, že zavolá metódu generate príslušného generátora a predá mu toto pole bytov. Po tom čo sa vygenerujú náhodné dáta je vygenerovaných extra 32 bytov, ktoré sa nastavia ako nový kľúč pseudonáhodného generátora. Keďže generátor Fortuna môže pracovať s rôznymi typmi pseudonáhodného generátora tak v tejto metóde nie je ošetrené maximálne množstvo dát, ktoré je možné vygenerovať pre jeden kľúč. Užívateľ využívajúci knihovňu aalg preto bude musieť vedieť s akým pseudonáhodným generátorom bude pracovať a musí preto vedieť ako často bude treba meniť vnútorný stav pseudonáhodného generátora a aké maximálne množstvo náhodných dát si môže dovoliť vygenerovať pre jeden kľúč.

```
public void generateRandomData(byte[] result){
   generator.generate(result);
   byte[] newKey = new byte[32];
   generator.generate(newKey);
   key = new ByteArraySymmetricKey(newKey);
   generator.setKey(key);
}
```

Treťou metódou je metóda reseed. Táto metóda slúži na zmenu vnútorného stavu pseudonáhodného generátora. Po zavolaní tejto metódy sa testujú podmienky, ktoré treba splniť na to aby mohol byť vnútorný stav pseudonáhodného generátora zmenený. Prvou podmienkou je to, že od posledného zavolania tejto metódy muselo uplynúť najmenej 100 ms. Na to využívame timer a jeho metódu enoughTimePassed. Ak podmienka nie je splnená tak metóda enouhgTimePassed vráti hodnotu false, ktorá je znegovaná a vykoná sa kód v bloku if, ktorý zaistí to, že vyskočíme z metódy reseed s návratovou hodnotou false.

```
if(!(timer.enoughTimePassed())){
   return false;
}
```

Druhou podmienkou je to, že sa muselo od naposledy uskutočneného reseedu nazbierať dostatok entropie. Robí sa to takým spôsobom, že sa zistí koľko entropie je nazbieranej v pooli 0 a toto číslo sa porovná s konštantou minEntropyInPool0. Táto konštanta má hodnotu 1000 a vyjadruje 1000 bytov. Keďže entropia sa medzi jednotlivé pooli prerozdeľuje rovnomerne a poolov je dokopy 32, tak to znamená, že reseed môžeme vykonať až po nazbieraní 32 kB dát entropie.

```
if(!(acumulator.getPool(0).getSize() > minEntropyInPool0)){
   return false;
}
```

Keď sú obe podmienky splnené dochádza k samotnému reseedu. Podľa teórie sa musí pool P_0 použiť v každom reseede, pool P_1 v každom druhom, pool P_2 v každom štvrtom atď. Pritom nový kľúč sa vypočíta zhešovaním starého kľúča zreťazeného s nahromadenou entropiou, viz 3.10. Pri implementácii využívame tzv. inkrementálne hešovanie. Pri tomto inkrementálnom hešovaní nemusíme počítať hešovaciu funkciu naraz, ale voláme metódu update a predávame hešovacej funkcii nazhromaždené dáta jednotlivo po pooloch. Hešovaciu funkciu nikdy nemažeme, to znamená, že si vždy pamätá starý kľúč, stačí nám len pridať entropiu z potrebných poolov. Potrebné pooli vyberáme pomocou algoritmu ktorý nám v smyčke for prechádza jednolivé pooli a pool P_q je zahrnutý vtedy ak čítač reseedCounter je deliteľom čísla 2^q.

```
for(int q = 0; q<32; q++){
   if(((2^q)%reseedCounter) == 0){
   hash.update(acumulator.getPool(q).getEntropy(), 0,
      acumulator.getPool(q).getEntropy().length);
   }
}
```

Po tom čo sme heš inkrementálne predpočítali tak zavoláme metódu finish, ktorá nám vráti výsledný heš. Tento heš je však 64 bitový, preto ho treba upraviť na veľkosť 32 bitov. Tento heš uložíme do poľa bytov z ktorého potom vytvoríme nový kľuč a nastavíme ním nový stav pseudonáhodného generátora.

```
byte[] keyBytes = Arrays.copyOfRange(hash.finish(), 0, 32);
```

Potom čo sme vypočítali nový kľúč tak už len inkrementujeme čítač reseedCounter, vyresetujeme timer a vrátime hodnotu true, ktorá indikuje, že zmena vnútorného stavu pseudonáhodného generátora bola vykonaná.

Poslednou štvrtou metódou triedy Fortuna je metóda finish. Keďže celý program beží vo viacerých vláknach tak je potrebné aby sa pri ukončení programu pouzatváral beh všetkých vlákien. V tele tejto metódy je for cyklus ktorý prebehne všetky zdroje entropie uložené v zozname etropySource a nastaví premennú running v týchto vláknach na false čím sa ukončí ich beh.

Trieda Timer

Pomocou objektu triedy Timer zisťujeme či prešiel dostatočne dlhý čas na to aby sme vykonali ďalší reseed. Táto trieda obsahuje dva atribúty a dve metódy. Vždy keď chceme vedieť či prešlo potrených 100 ms na vykonanie ďalšieho reseedu tak zavoláme metódu enoughTimePassed. V tejto metóde sa do atribútu actualTime uloží aktuálny čas pomocou metódy System.currentTimeMillis(). Tento čas je vyjadrený v milisekundách ktoré ubehli od roku 1970. Túto hodnotu odčítame následne od referenčnej hodnoty statTime a ak je rozdiel väčší ako 100 tak metóda vráti pravdivostnú hodnotu true. Druhou metódou je metóda reset. Táto metóda nastaví atribút startTime na aktuálny referenčný čas.

Ostatné triedy

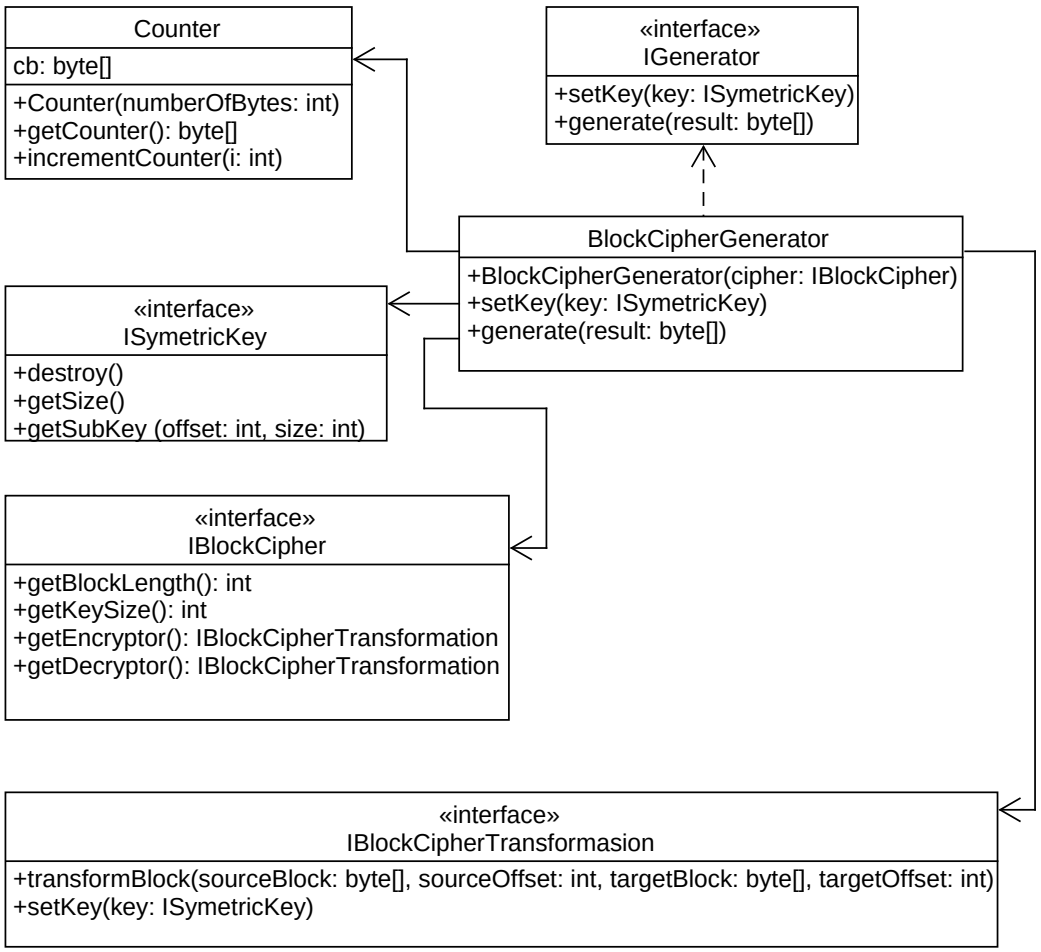
Generátor Fortuna ďalej pracuje s rozhraniami IGenerator, IHashFunction, ISymmetricKey a triedami EntropySource a Acumulator. Tieto triedy a rozhrania zaisťujú generátoru Fortuna vysokú modularitu a dovoľujú mu použiť rôzne typy pseudonáhodných generátorov, hešov a zdrojov entropie. Tieto bloky z ktorých je poskladaný generátor Fortuna môžeme chápať ako akési podsystémy, ktoré sú detailne popísané v jednotlivých podkapitolách.

4.2 Subsystém Generator

Na obrázku 4.2 môžeme vidieť diagram tried pre subsystém generátor. Tento subsystém je konkrétnou implementáciou pre blokové šifry. Na obrázku môžeme vidieť štyri rozhrania a dve triedy, ktoré budú detailnejšie opísané v jednotlivých podkapitolách.

Rozhranie IGenerator

Jeden z atribútov triedy Fortuna je ukazateľ IGenerator. Na to aby sme mohli vytvoriť pseudonáhodný generátor, ktorý by vedel pracovať v generátore Fortuna musíme



Obr. 4.2: Diagram tried pre subsystém Generator.

vytvoriť triedu, ktorá implementuje rozhranie IGenerator. Toto rozhranie obsahuje dve metódy. Prvou je metóda setKey pomocou ktorej meníme kľúč alebo vnútorný stav generátora, druhou metódou je metóda generate pomocou ktorej generujeme náhodné dáta.

BlockCipherGenerator

BlockCipherGenerator je už konkrétna implementácia pseudonáhodného generátora. Táto trieda implementuje rozhranie IGenerator. Medzi atribúty tejto triedy patria counter triedy Counter, key rozhrania ISymetricKey, cipher rozhrania IBlockCipher a encryptor rozhrania IBlockCipherTransformation. Pri vytváraní objektu z triedy BlockCipherGenerator vkladáme do konštruktoru šifru s ktorou budeme pracovať, viz kód nižšie v ktorom používame 256 bitovú šifru AES z knižnice aalg.

```
BlockCipherGenerator aesCipherGenerator = new BlockCipherGenerator(new
    AesCipher(32));
```

V konštruktore sa ďalej inicializuje objekt counter v závislosti na dĺžke šifry. Ďalej sa v ňom v závislosti od dĺžky šifry inicializuje pole bajtov obsahujúce samé nuly z ktorého vytvorí symetrický kľúč. Zo šifry sa ďalej odvodí encryptor, ktorému je predaný tento kľúč.

Trieda BlockCipherGenerator ďalej implementuje dve metódy. Prvou je metóda setKey pomocou ktorej predáme kľúč encryptoru. Druhou metódou je metóda generate pomocou ktorej generujeme náhodné dáta. Do parametru tejto metódy predávame pole bytov, ktoré má byť naplnené náhodnými dátami. Bloková šifra ako už vychádza z názvu však dokáže naraz zakódovať len obmedzené množstvo dát, tzv. blok. V našom prípade u šifry AES pri generácii dát šifrujeme 16 bytový čítač. Ak preto potrebujeme vygenerovať napríklad 100 bytov náhodných dát. Musíme v našom prípade vygenerovať 7 blokov a výstup orezať na potrených 100 bytov.

V zobrazenom kóde môžeme vidieť presnú implementáciu. Na začiatku vytvoríme konštantu blockLength, ktorá reprezentuje veľkosť bloku s akou pracuje bloková šifra. Ďalej definujeme pole bytov o veľkosti tohto bloku a premennú offset. V cykle while potom generujeme jednotlivé bloky a priraďujeme ich do poľa result až kým nie je úplne naplnené. Na generáciu bloku náhodných dát voláme encryptor a jeho metódu transformBlock, ktorej do parametru predáme aktuálnu hodnotu čítača, vstupný offset, pole bytov block a výstupný offset. Akonáhle vygenerujeme block tak inkrementujeme čítač. V ďalšom krokom zisťujeme koľko bytov z jedného bloku máme priradiť do výsledného pola result. Na to využijeme statickú metódu min triedy Math. Do parametru predáme počet bytov, ktoré ešte treba vygenerovať a veľkosť jedného bloku. Ak metóda min vyhodnotí hodnotu premennej blockLengh ako menšiu, tak sa celý blok nakopíruje do poľa result a celý proces sa opakuje v ďalšom cykle kedy je vygenerovaný ďalší blok dát. Ak metóda min vyhodnotí, že počet bytov ktoré je ešte potrebné vygenerovať je menší ako počet bytov v jednom bloku tak sa tento zvyšní počet bytov nakopíruje do pola result a cyklus while sa ukončí lebo premenná offset sa bude rovnať veľkosti pola result.

```
public void generate(byte[] result){
   final int blockLength = encryptor.getCipher().getBlockLength();
   byte[] block = new byte[blockLength];
   int offset = 0;

   while(offset<result.length){
      encryptor.transformBlock(counter.getCounter(), 0, block, 0);
```

```
      counter.incrementCounter(0);

      final int size = Math.min(result.length-offset, blockLength);
      System.arraycopy(block, 0, result, offset, size);
      offset += size;
   }
}
```

Trieda Counter

Na to aby sme šifru AES mohli použiť na generovanie náhodných dát potrebujeme 128 bitový čítač. Túto 128 bitovú hodnotu je potrené predávať encryptoru vo forme pola bytov.

Pri vytváraní objektu triedy counter predáme konštruktoru veľkosť bloku s ktorou bude bloková šifra pracovať zavolaním metódy cipher.getBlockLength. Podľa toho sa vytvorí aj veľkosť čítača, v našom prípade to bude hodnota 16 reprezentujúca 16 bytové pole, všetky byty poľa sa potom nastavia na hodnotu -128. Je to z toho dôvodu, že Java nepodporuje neznamienkový byte, tzv. unsigned byte, ktorý by nadobúdal hodnoty 0-255. Trieda Counter ďalej obsahuje dve metódy. Prvá metóda getCounter vracia hodnotu čítača vo forme pola bytov. Druhá metóda incrementCounter slúži na inkrementáciu čítača.

V nižšie zobrazenom kóde môžeme vidieť, že inkrementácia je vyriešená rekurzívnou funkciou. Keď chce vonkajší zdroj, teda generátor inkrementovať čítač tak vždy zavolá funkciu v tvare incrementCounter(0). Tým povie, že chce inkrementovať nultý byte čítača. Ak je už v nultom byte nastavená hodnota 127 a hrozí jeho pretečenie tak sa mu nastaví hodnota -128 a znova sa zavolá metóda incrementCounter(1) tentokrát s požiadavkou na inkrementáciu prvého bytu čítača atď.

Keďže je čítač 128 bitový, skladá sa teda zo 16-tych bytov s indexáciou 0-15. Preto je na začiatku metódu podmienka, ktorá zaistí, že po pretečení celého čítača je inkrementácia ukončená a začína sa znova z východzieho stavu. Pretečenie celého čítača však nastáva len najskôr za každých 13,5 roka.

```
public void incrementCounter(int i){
   if(i==cb.length){
      return;
    }
    if(cb[i] == 127){
        cb[i] = -128;
        incrementCounter(i+1);
    }else{
```

```
        ++cb[i];
    }
}
```

Rozhranie ISymetricKey

Jedná sa o rozhranie symetrického kľúča knihovny aalg. Toto rozhranie sa používa pri nastavovaní vnútorného stavu pseudonáhodného generátora.

Rozhranie IBlockCipher

Pri vytváraní objektu BlockCipherGenerator predávame do parametru blokovú šifru s ktorou bude tento generátor pracovať. V našom prípade sme využili šifru AES z knihovny aalg. Môžeme však využiť hociktorú blokovú šifru. Stačí ak spomínaná šifra implementuje rozhranie IBlockCipher a všetky príslušné metódy ako getBlock-Length, getKeysize, getEncryptor a getDecryptor.

Rozhranie IBlockCipherTransformation

Rozhranie hnihovny aalg používané pri šifrovaní pomocou blokových šifier.

4.3 Subsystém Entropy Sources

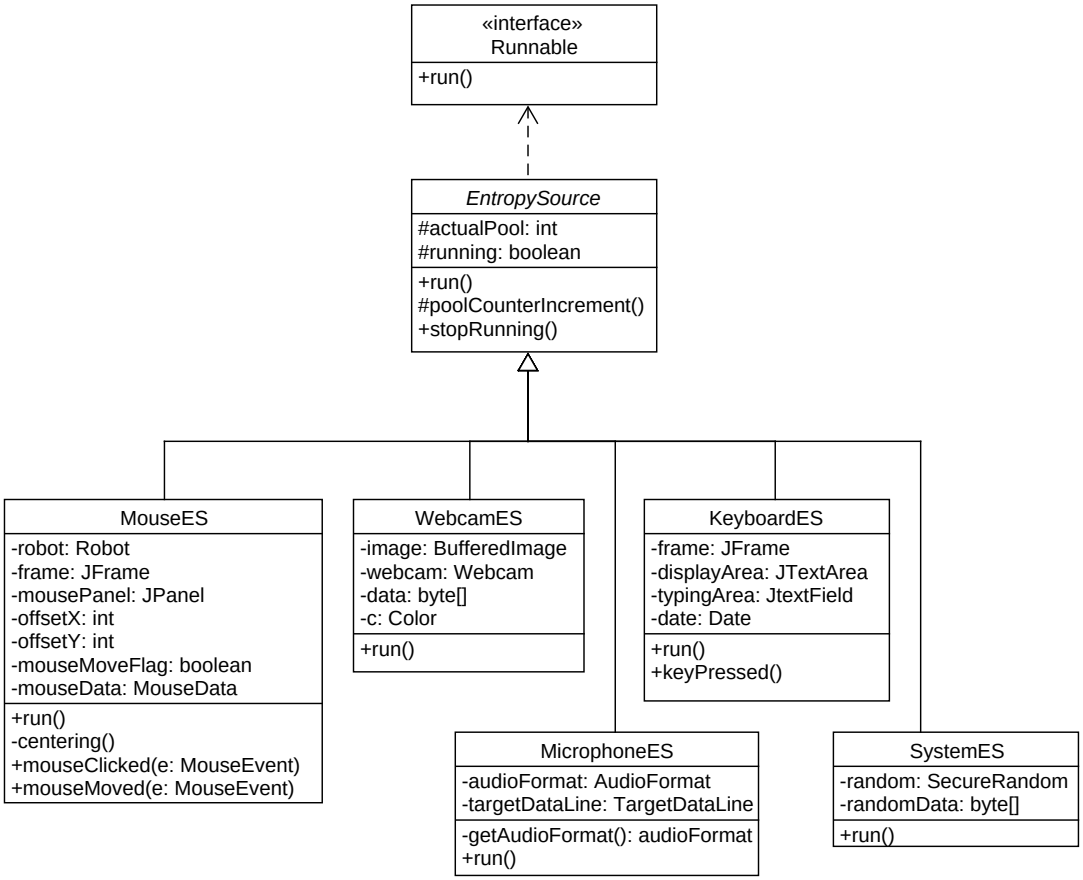
Generátoru Fortuna môžeme priradiť toľko zdrojov entropie koľko len chceme. Stačí mu len predať objekt triedy EntropySource. Na obrázku 4.3 môžeme vidieť ako sú zdroje entropie do generátoru Fortuna začleňované.

Rozhranie Runnable

Každý zdroj entropie je navrhnutý tak aby bežal vo vlastnom vlákne. Vďaka tomu môžeme kedykoľvek zdroje entropie do generátoru Fortuna pridávať alebo odoberať. Takisto pád vlákna napríklad webkamery neohrozí beh programu a zdroje entropie môžu takto neustále zbierať entropiu bez ohľadu na to či pseudonáhodný generátor generuje dáta alebo nie. Z tohto dôvodu každý zdroj entropie implementuje rozhranie Runnable.

Trieda EntopySource

Trieda EntropySource je abstraktná trieda implementujúca rozhranie Runnable. Táto trieda obsahuje dva atribúty a tri metódy. Prvým atribútom je actualPool,



Obr. 4.3: Diagram tried pre subsystém Entropy Sources.

ktorý je reprezentovaný dekadickou hodnotou a určuje to do ktorého poolu v akumulátore sa bude nazbieraná entropie ukladať. S týmto atribútom súvisí aj metóda poolCounterIncrement, ktorá tento counter vždy inkrementuje v rozsahu 0-31.

Druhým atribútom je premenná running v ktorej je uložená logická hodnota určujúca či má vlákno bežať a kedy má skončiť. Pri spustení vlákna sa tento atribút nastaví na hodnotu true. Akonáhle je však atribút nastavený na false, vlákno sa ukončí. Tento atribút ma modifikátor volatile.

```
protected volatile boolean running;
```

Modifikátor volatile sa v Jave používa vtedy ak hodnota atribútu môže byť zmenená iným vláknom. S týmto súvisí aj metóda stopRunning. Ak má byť napríklad beh generátoru Fortuna ukončený tak Fortuna zavolá vo všetkých zdrojoch entropie metódu stopRunning, ktorá nastaví atribút running na false čím ukončí činnosť týchto vlákien.

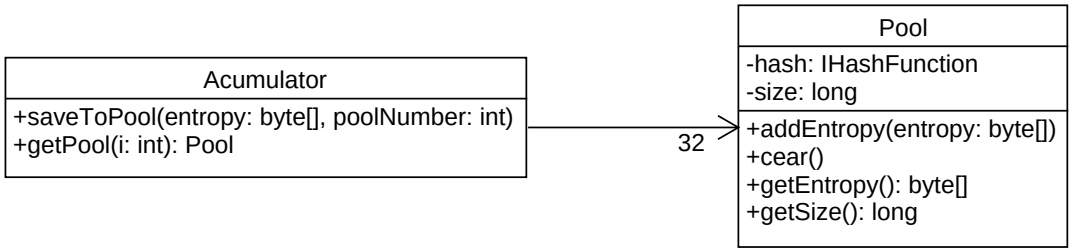
Poslednou metódou je metóda run. V metóde run je smyčka while ktorá testuje premennú running. Vo vnútri tejto smyčky je umiestnený kód špecifický pre každý zdroj entropie, ktorý sa stará o zber entropie.

Ostatné triedy

Ostatné triedy ako MouseES, MicrophoneES, WebcamES, SystemES a KeyboardES sú už konkrétnymi zdrojmi entropie. Funkčnosť aj implementácia týchto zdrojov entropie spolu s ich analýzou už bola v tomto texte popísaná. Pri integrácii do knihovny aalg boli tieto triedy len mierne poupravené a to takým spôsobom, že boli zmazané prípadne pozmenené metódy na výpočet a zobrazovanie medzivýsledkov. Ďalšou úpravou týchto tried je to, že zároveň náhodné dáta zbierajú a rovno aj spracovávajú s použitím bufferu.

4.4 Subsystém Akumulátor

Subsystém Akumulátor a jeho diagram tried je vyobrazený na obrázku 4.4. Tento subsystém je zo všetkých najjednoduchší a skladá sa len z dvoch tried.



Obr. 4.4: Diagram tried pre subsystém Akumulátor

Trieda Acumulator

Trieda Akumulátor je pevnou súčasťou generátoru Fortuna a je to práve táto trieda, ktorá robí generátor Fortuna odolný proti útokom zvonku. Z implementačného hľadiska táto trieda obsahu poľe poolov o veľkosti 32 a dve metódy.

SaveToPool je metóda ktorá zapisuje dáta do poolu, pri volaní tejto metódy jej konkrétny zdroj entropie predá pole bytov náhodných dát ľubovoľnej dĺžky a dekadickú hodnotu poolu do ktorého si praje tieto dáta zapísať. Druhá metóda getPool zase vráti potrebný pool keď sa generátor Fortuna snaží extrahovať dáta z poolu.

Objekt triedy Acumulator sa inicializuje vždy keď sa vytvára objekt triedy Fortuna. Pri inicializácii objektu Acumulator je v jeho konštruktore rovnako inicializovaných 32 objektov triedy pool. Na inicializáciu jednotlivých objektov triedy pool využijeme funkciu Java zvanú reflexia. Týmto spôsobom môžeme pri vytváraní objektu z triedy Acumulator predať konštruktoru hešovaciu funkciu ktorou sa bude hešovať.

```
Acumulator acumulator = new Acumulator(Sha512.class);
```

Následne sa v konštruktore vytvorí jedinečných 32 instancií tejto hešovacej funkcie a tieto instancie sú predané jednotlivým poolom.

```
public Acumulator(final Class<? extends IHashFunction> hashClass) {
   for (int i = 0; i < pool.length; i++) {
      try {
         final IHashFunction hash = hashClass.newInstance();
         pool[i] = new Pool(hash);
      }
      catch (InstantiationException | IllegalAccessException ex) {
         throw new RuntimeException("Cannot create hash function", ex);
      }
   }
}
```

Trieda Pool

Pri ukladaní entropie do poolov treba z implementačného hľadiska myslieť na pamäťové a výpočetné nároky programu. Z teórie vyplýva, že zbieraná entropia sa medzi pooli distribuje rovnomerne a že napríklad pool P_{31} sa použije pri reseede každých 14 rokov. Nie je preto veľmi vhodné aby sme zahlcovali pamäť dátami, ktoré pravdepodobne nikdy nepoužijeme. Na druhej strane je tiež veľmi náročné vypočítať heš z veľkého objemu dát, ktoré program zbieral dlhšiu dobu. Riešením je opäť použitie inkrementálneho hešovania.

Trieda Pool preto obsahuje dva atribúty, prvým je hešovacia funkcia a druhým je premenná size, ktorá nám udáva koľko dát sme už v poole nazbierali. Pri ukladaní dát do poolu sa volá metóda addEntropy, ktorej do parametru predáme pole bytov ľubovolnej veľkosti. Vnútri tejto metódy je potom z objektu hešovacie funkcie volaná metóda update, ktorej predáme toto pole bytov. Metóda update nám inkrementálne vypočítava heš a vďaka nej sú pamäťové nároky konštantné a eliminujú sa týmto spôsobom aj výkonové špičky. Keďže do poolu týmto spôsobom môžu ukladať dáta

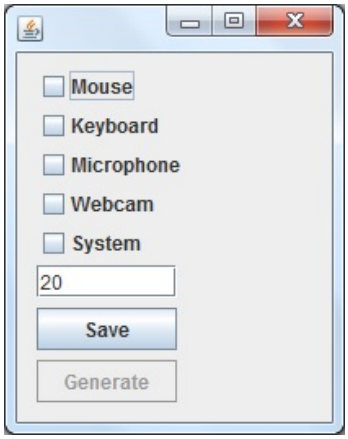
rôzne zdroje entropie, inak povedané vlákna a taktiež nie je vylúčené, že sa o to môžu pokúšať zároveň, tak je potrebné na túto metódu upevniť zámok, teda ju synchronizovať kľúčovým slovom synchronized. Okrem toho, že sme zavolali metódu update treba ešte aj pripočítať veľkosť pola k atribútu size aby sme vedeli koĺko dát už bolo do poolu uložených.

S týmto atribútom size súvisí ďalšia metóda getSize, ktorá tento atribút vracia. Túto metódu využíva trieda Fortuna, ktorá sa dotazuje poolu P_0 na objem nazbieraných dát a zisťuje tak či je možné vykonať ďalší reseed.

Keď je potrebné dáta z poolu extrahovať tak sa zavolá metóda getEntropy. V tejto metóde sa pomocou hešovacej funkcie volaním metódy finish vypočíta heš reprezentujúci všetky nazbierané dáta v poole. Tento heš je reprezentovaný polom bytov a tvorí návratovú hodnotu funkcie. Okrem toho sa v tejto metóde volá ešte aj metóda clear. Táto metóda má za úlohu vynulovať atribút size a takisto nanovo inicializovať hešovaciu funkciu volaním metódy initialize.

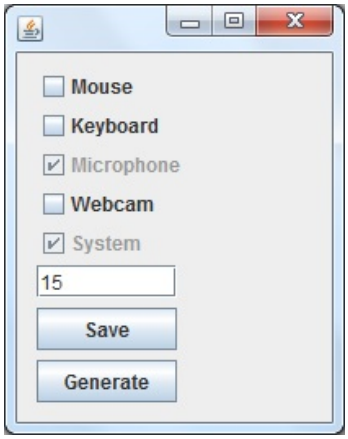
5 VZOROVÝ PROGRAM

Po tom čo bol generátor náhodných čísel Fortuna implementovaný do knihovny aalg bol vytvorený aj vzorový program, ktorý využíva túto knihovnu na generáciu náhodných dát. Na obrázku 5.1 môžeme vidieť hlavné okno programu. Ako môžeme vidieť tlačítko generate je v tomto stave zablokované, najprv musíme zakliknúť zdroje entropie, ktoré si prajeme použiť. Akonáhle zdroj entropie zaškrtneme, je okamžite spustený a začína zbierať entropiu.



Obr. 5.1: Okno vzorového programu.

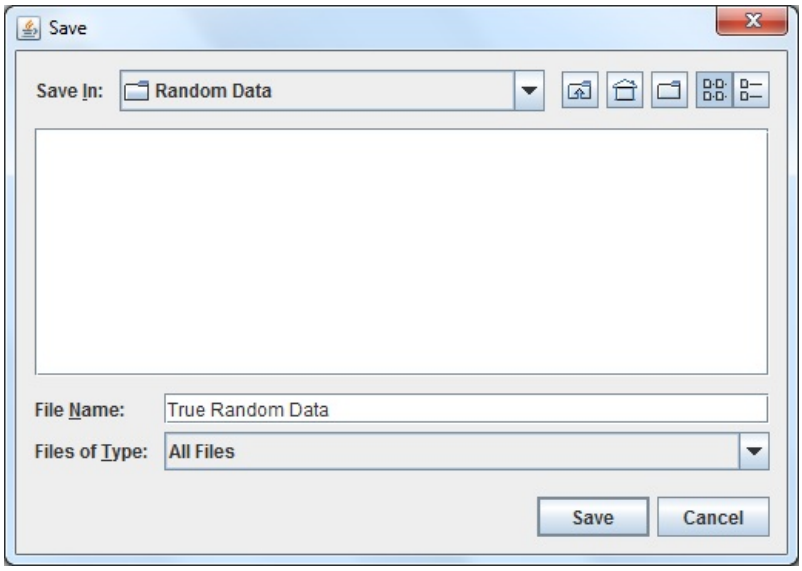
Na obrázku 5.2 už môžeme vidieť, že zdroje entropie nazbierali dostatok entropie a tlačítko generate sa tým pádom aktivovalo a my môžeme generovať náhodné dáta. Predtým však ešte môžeme zadať koľko bytov dát si prajeme vygenerovať a takisto môžeme kliknúť na tlačítko save a definovať kde chceme dáta uložiť.



Obr. 5.2: Okno vzorového programu po nazbieraní dostatku entropie.

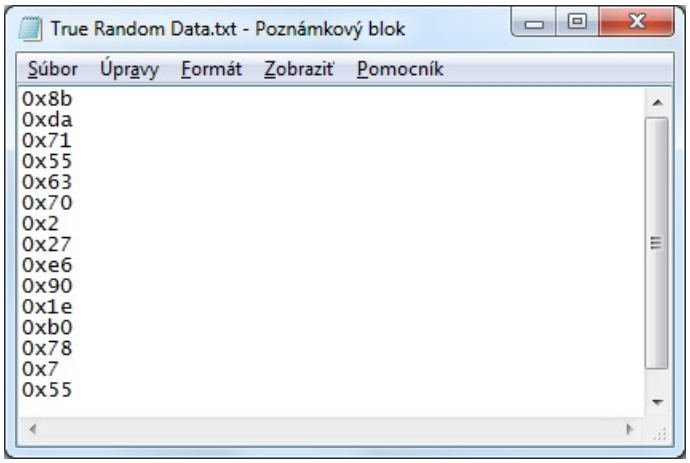
Ukážku môžeme vidieť na obrázku 5.3 kde sme vytvorili súbor True Random Data v adresári Random Data, týmto spôsobom sa nám v adresári vytvorí textový

súbor s patričným menom a do tohto súboru sa potom budú generované náhodné dáta zapisovať. Ak na tlačítko save neklikneme a nešpecifikujeme kam sa majú dáta ukladať, tak sa budú ukladať do aktuálneho adresára v ktorom sa nachádza aj program pod názvom Cryptodata.



Obr. 5.3: Okno na uloženie vygenerovaných dát.

Na obrázku 5.4 môžeme vidieť dáta, ktoré sa boli vygenerované a uložené do súboru True Random Data. Jednotlivé byty sú zapísané v hexadecimálnej forme a môžu byť načítané inou aplikáciou.



Obr. 5.4: Vygenerované dáta uložené v textovom súbore.

6 ZÁVER

Táto práca sa zaoberala generovaním náhodných čísel, teoretickým opisom generátoru náhodných čísel Fortuna, posúdením a zhodnotením rôznych zdrojov entropie a implementáciou generátoru Fortuna do knihovny aalg.

Z interaktívnych zdrojov entropie boli vybrané metódy pohybu myšou a písania na klávesnici. Z neinteraktívnych zdrojov boli vybrané metódy získavania šumu mikrofónu a šumu webkamery. Zo systémových zdrojov bol vybraný implementovaný kryptografický generátor v Jave. Z hľadiska náhodnosti dát najlepšie obstáli neinteraktívne zdroje entropie, teda šum mikrofónu a šum webkamery. Interaktívne zdroje majú v tomto ohľade limitujúce faktory. Čo sa týka pohybu myšou tak získané dáta nemajú uniformné rozloženie. Náhodnosť dát ďalej negatívne ovplyvňuje nastavenie citlivosti myši. V prípade písaní na klávesnici je zase náhodnosť dát negatívne ovplyvnená pravidelnosťou s akou užívateľ stláča klávesy. Z hľadiska objemu získaných dát opäť majú navrch neinteraktívne zdroje entropie. V prípade zachytávania šumu pomocou mikrofónu je objem získaných dát závislý od vzorkovacieho kmitočtu, pri vzorkovacom kmitočte 192 kHz sme schopný získavať náhodné dáta rýchlosťou 24 kB/s. V prípade zachytávania šumu pomocou webkamery zase objem získaných dát ovplyvňuje výpočetný výkon a to ako rýchlo stíhame spracovávať zachytené obrázky. V našom prípade bola rýchlosť 221,54 kB/s. Čo sa týka interaktívnych metód tak tam sa objem získaných dát za čas posudzuje pomerne ťažko pretože sú závislé od používateľa. Napriek tomu ak vezmeme do úvahy fakt, že jedným stisknutím klávesy vygenerujeme 1 B dát, tak užívateľ by musel stisknúť kvantum kláves za sekundu aby sa vyrovnal neinteraktívnym metódam. Čo sa týka pohybu myši tak tam sme schopný pri kontinuálnom pohybe zachytiť 735 dekadických hodnôt za sekundu. Aj keby sme každú hodnotu vyjadrili v 8 bitovom formáte, tak aj napriek tomu by sme boli schopný dostať sa len na rýchlosť 0,735 kB/s. Z hľadiska náhodnosti aj objemu získaných dát sú neinteraktívne zdroje entropie lepšie ako interaktívne zdroje entropie. Z hľadiska bezpečnosti sa však pri implementácii generátoru Fortuna snažíme použiť všetky zdroje entropie aby sme zabezpečili čo najväčšiu redundanciu systému.

Implementácia generátoru Fortuna do knihovny aalg bola úspešná. Celá aplikácia je navrhnutá modulárne a je možné použiť rôzne typy pseudonáhodných generátorov, hešov a zdrojov entropie. Implementácia bola otestovaná na vzorovom programe, ktorý pracuje s blokovou šifrou AES, používa hešovaciu funkciu SHA a vyššie spomínané zdroje entropie. Použitá šifra AES-256 aj heš SHA512 sú taktiež súčasťou knihovny aalg.

LITERATÚRA

[1] BALDWIN, Richard. *Java Sound, Getting Started, Part 2, Capture Using Specified Mixer* [online]. [cit. 1. 12. 2014] Dostupné z URL: <http://www.developer.com/java/other/article.php/1579071/Java-Sound-Getting-Started-Part-2-Capture-Using-Specified-Mixer.htm>.

[2] BURDA, Karel. *Bezpecnost informacních systému*. 2013, posledná aktualizácia 1. 10. 2013 [cit. 12. 4. 2015]

[3] CLEWETT, James. In: *Random Numbers - Numberphile* [online]. 2011 [cit. 1. 12. 2014] Dostupné z URL: <https://www.youtube.com/watch?v=SxP30euw3-0>.

[4] GOODIN, Dan. *"We cannot trust" Intel and Via's chip-based crypto, FreeBSD developers say* [online]. [cit. 1. 12. 2014] Dostupné z URL: <http://arstechnica.com/security/2013/12/we-cannot-trust-intel>.

[5] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications* . 2010 [cit. 1. 12. 2014] ISBN: 978-0-470-47424-2

[6] MECHALAS, John. *Intel® Digital Random Number Generator (DRNG) Software Implementation Guide* [online]. posledná aktualizácia 15. 5. 2014 [cit. 1. 12. 2014] Dostupné z URL: <https://software.intel.com/en-us/articles/intel-digital-random-number-generator>.

[7] JAVA Documentation, sound. [online]. Dostupné z URL: <https://docs.oracle.com/javase/tutorial/sound/capturing.html>.

[8] HAAHR, Mads. *Introduction to Randomness and Random Numbers* [online]. [cit. 1. 12. 2014] Dostupné z URL: <http://www.random.org/randomness/>.

[9] Projekt sarxos. [online]. Dostupné z URL: <https://github.com/sarxos/webcam-capture>.

[10] Sullivan, Nick. *Ensuring Randomness with Linux's Random Number Generator* [online]. posledná aktualizácia 3. 10. 2013 [cit. 1. 12. 2014] Dostupné z URL: <http://blog.cloudflare.com/ensuring-randomness-with-linuxs-random-number-generator/>.

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

AES	Advanced Encryption System
API	Application Programming Interface
CCD	Charge-coupled device
CTR	Counter
LFSR	Linear Feedback Shift Register
PCM	Pulse Code Modulation
PRNG	Pseudo Random Number Generator
RGB	Red, Green, Blue
SHA	Secure Hash Algorithm
SSL	Secure Socket Layer
TRNG	True Random Number Generator
UML	Unified Modeling Language

A OBSAH PRILOŽENÉHO CD

- Elektronická verzia diplomovej práce.
- Zdrojové kódy knihovny aalg.
- Zdrojové kódy programov použitých na analýzu zdrojov entropie.
- Textové súbory obsahujúce náhodné dáta, ktoré boli analyzované v texte.