Masaryk University
Faculty of Informatics

# Automated task management for BOINC infrastructure and EACirc project

Bachelor Thesis

**Ľubomír Obrátil**

Brno, Spring 2015

# Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ľubomír Obrátil

**Advisor:** RNDr. Petr Švenda, Ph.D.

# Acknowledgement

I would like to express my gratitude to Pert Švenda for his guidance, inspiration and valuable feedback. My further thanks goes to Martin Ukrop and other lab members for their help and advice. Also, I'd like to thank to my girlfriend for her continuous support and endless patience.

# Abstract

This thesis proposes a way to automate generation and processing of experiments for EACirc application running in BOINC infrastructure. Developed tool, Oneclick, is tested by recreating EACirc experiments on wide range of cryptographic functions. Obtained results are compared to results of older experiments. Comparison is used to detect significant changes caused by modifications in EACirc implementation. Alternative ways to test data stream for randomness are presented and three chosen test suites are compared.

# Keywords

automation, experiments recomputation, data stream randomness, statistical test suites, EACirc, CRoCS, BOINC, eStream, SHA-3

# Contents

1 **Introduction** . . . 3
2 **Automation of EACirc** . . . 4
  2.1 *Terminology* . . . 4
  2.2 *Motivation* . . . 5
  2.3 *Oneclick design* . . . 5
3 **Implementation of Oneclick** . . . 8
  3.1 *Generation of files* . . . 8
    3.1.1 Oneclick options . . . 8
    3.1.2 Configuration files generation . . . 10
    3.1.3 Script files generation . . . 10
  3.2 *Creation of jobs on BOINC server* . . . 10
    3.2.1 Creation of single batch . . . 10
    3.2.2 Automated creation of multiple batches . . . 10
    3.2.3 Authentication . . . 11
  3.3 *Download of results from BOINC server* . . . 11
    3.3.1 Downloading results of single workunit . . . 11
    3.3.2 Assimilator . . . 11
    3.3.3 Downloading results of single batch . . . 11
    3.3.4 Comparison of download times . . . 12
    3.3.5 Automated download of multiple batches . . . 12
  3.4 *Result processing* . . . 12
    3.4.1 Consistency and error checks . . . 12
    3.4.2 Processing of single batch . . . 13
    3.4.3 Post-processors . . . 13
    3.4.4 Processing of multiple batches . . . 13
4 **Verification of previous computations** . . . 15
  4.1 *EACirc settings and output* . . . 15
    4.1.1 Result format . . . 15
  4.2 *Random data distinguishing* . . . 16
  4.3 *Oneclick settings* . . . 17
    4.3.1 eStream project settings . . . 17
    4.3.2 SHA3 project settings . . . 18
  4.4 *Oneclick performance* . . . 18
  4.5 *Comparison of results of experiments* . . . 19
    4.5.1 eStream project results . . . 19
    4.5.2 SHA3 project results . . . 22
  4.6 *Conclusions based on comparison* . . . 25
5 **Alternative statistical test suites overview** . . . 26
  5.1 *Statistical test suites* . . . 26
    5.1.1 NIST Statistical Test Suite . . . 26
    5.1.2 DIEHARD . . . 27
    5.1.3 DIEHARDER . . . 28
    5.1.4 TestU01 . . . 28

5.1.5 PractRand . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.1.6 RaBiGeTe . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.1.7 CryptoStat . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.1.8 YAARX . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.2 *Comparison of chosen test suites* . . . . . . . . . . . . . . . 31
5.2.1 Comparison results . . . . . . . . . . . . . . . . . . . . . 32
6 **Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
A **Data attachment** . . . . . . . . . . . . . . . . . . . . . . . . . 37

# 1 Introduction

Automation is an approach, which purpose is to reduce or completely eliminate human interaction with manufacture of products, management of production, testing and other processes. Automation process has begun to occur from year 1920 with invention of automatic production lines and from 1950 it was being implemented into modern factories. Main aim of automation is to increase efficiency, save manpower, resources and time, improve quality of products and eliminate errors made by humans.

EACirc [Š+12] is a project running under CRoCS (Centre for Research on Cryptography and Security) laboratory [Cen] on Faculty of Informatics, Masaryk University. Currently, EACirc is used for distinguishing of outputs of cryptographic functions and truly random data. For this purpose, it uses hardware-like circuits emulated by software and generated using evolutionary algorithms. Considering long duration of this process, project was modified and is running on BOINC (Berkeley Open Infrastructure for Network Computing). BOINC [Uni] is infrastructure for distribution of work units assigned by server between its clients and subsequent collection of results.

Aim of the thesis was to create a tool usable for automatic generation of work units for EACirc and processing of their results. Manually, this process can take nontrivial amount of time, but with help of developed tool should be done in matter of minutes with minimal user interaction. Tool should be extensible, usable on Windows and Linux systems and well documented. User documentation will be published on wiki of EACirc project. Actual usability of tool will be tested on automated recomputing of old EACirc experiments.

Chapter 2 closely describes motivation for this tool, terminology used in thesis, design of the tool and provides simplified work-flow of proposed solution. Implementation of individual parts of the tool is in detail described in Chapter 3.

In Chapter 4 we describe process of recomputation of old experiments. Chapter contains settings used for EACirc and our tool. We also present duration of tool's operations and comparison and interpretation of results of older experiments and results of experiments executed by application.

Chapter 5 overviews some of the alternative approaches to randomness testing. From mentioned statistical test suites we have chosen three and compared their ability to detect non-randomness in various cryptographic functions' outputs.

# 2 Automation of EACirc

## 2.1 Terminology

Throughout this thesis, we will be using specific terms referring to BOINC and EACirc environment. Most common of them are listed here:

**(EACirc) project**
All implemented cryptographic functions in EACirc are subset of project. Project usually implements set of candidate functions of crypto competition e.g. eStream [Eur05] or SHA3 [Nat07].

**Algorithm**
One of the settings of EACirc. Represents tested cryptographic function. At this moment more than `20` functions are implemented for each EACirc project.

**Rounds**
Some algorithms can be limited by numbers of rounds used. EACirc is aimed mostly on testing algorithms that can be limited, because it's easier to spot non-randomness in outputs of weakened algorithms.

**(EACirc) job or workunit**
Smallest unit of computation. Refers to single run of EACirc application with specific settings as are algorithm, used project or number of rounds. One job takes usually from `10` minutes up to `2` hours to complete on BOINC client computer. Number of possible parallel jobs depends on number of client computers involved in computing.

**Batch**
Set of jobs with same settings i.e. clones. EACirc relies on statistics, so one job and it's results has no value. For successful results interpretation there needs to be multiple cloned jobs in batch (usually from `100` up to `1000` jobs).

**Batch results**
Statistics made from results of all jobs in batch.

**Result processing**
Getting batch results from files created by single jobs in batch.

**(BOINC) server**
Part of BOINC infrastructure. Server accepts requests for job creation and handles distribution of jobs among the clients. After computation, clients send results back to server.

**Evaluator**
Module in EACirc application. Evaluator is used in circuit evolution where it calculates circuit's fitness based on its output. Circuits with best fitness survive current generation and continue into the next one [Str02].

**Distinguisher**
Circuit generated by EACirc application used for distinguishing random data from non-random. Strong distinguisher has high success rate, where its success rate is ratio of correctly indicated non-random data streams and all data streams.

## 2.2 Motivation

Process of creating single batch with specific settings on BOINC server, using web interface is quite simple and fast. Apart from that we need to have prepared configuration file with settings for EACirc computation. Number of clones in batch is also specified on and handled by server. Whole action can be completed in around two minutes.

However, there is often need for multiple batches, with little differences in settings as algorithms or rounds. That means we need to create and manage multiple configuration files, one for each batch. If this process should be done manually, the risk of error in file making as well as time needed for job creation rapidly increases.

Another time consuming task is result processing. Each job in batch creates multiple files that have to be downloaded from BOINC server and then checked for errors, warnings and processed for results. For numerous batches, number of files can be in orders of thousands.

Most of these tasks can be automated so we decided to develop tool that would do this work instead of researchers, so that they can focus on result interpretation rather than result collecting. Scripts for result processing and downloading were written before as part of Martin Ukrop's thesis [Ukr13] but they weren't fully automating the whole process.

Purpose of this project was to develop multi-platform, robust and easy-to-use set of tools that would automatize tasks from workunit creation to result processing with human interaction as low as possible, hence the name Oneclick.

## 2.3 Oneclick design

We decided to use approach, where most of the work with files generation and result processing handles local machine. Our motivation for this option was better modifiability and debugging of local application rather than service running on BOINC server. Another advantage is that user can influence whole process and adjust it to his/her liking and preferences. Disadvantage is that user needs to have Oneclick tool on his computer and numerous result files have to be downloaded to machine before result processing.

Alternative to this approach is to control automation through web interface, where all of the work is done on the server. This option doesn't require user to have application but also can't be easily modified.

After discussion with members of CRoCS about different possible approaches, we broke down automation process into four main steps:

- **Generation of files**
- **Creation of jobs on BOINC server**
- **Download of results from BOINC server**
- **Result processing**

For Oneclick work-flow, see Figure 2.1

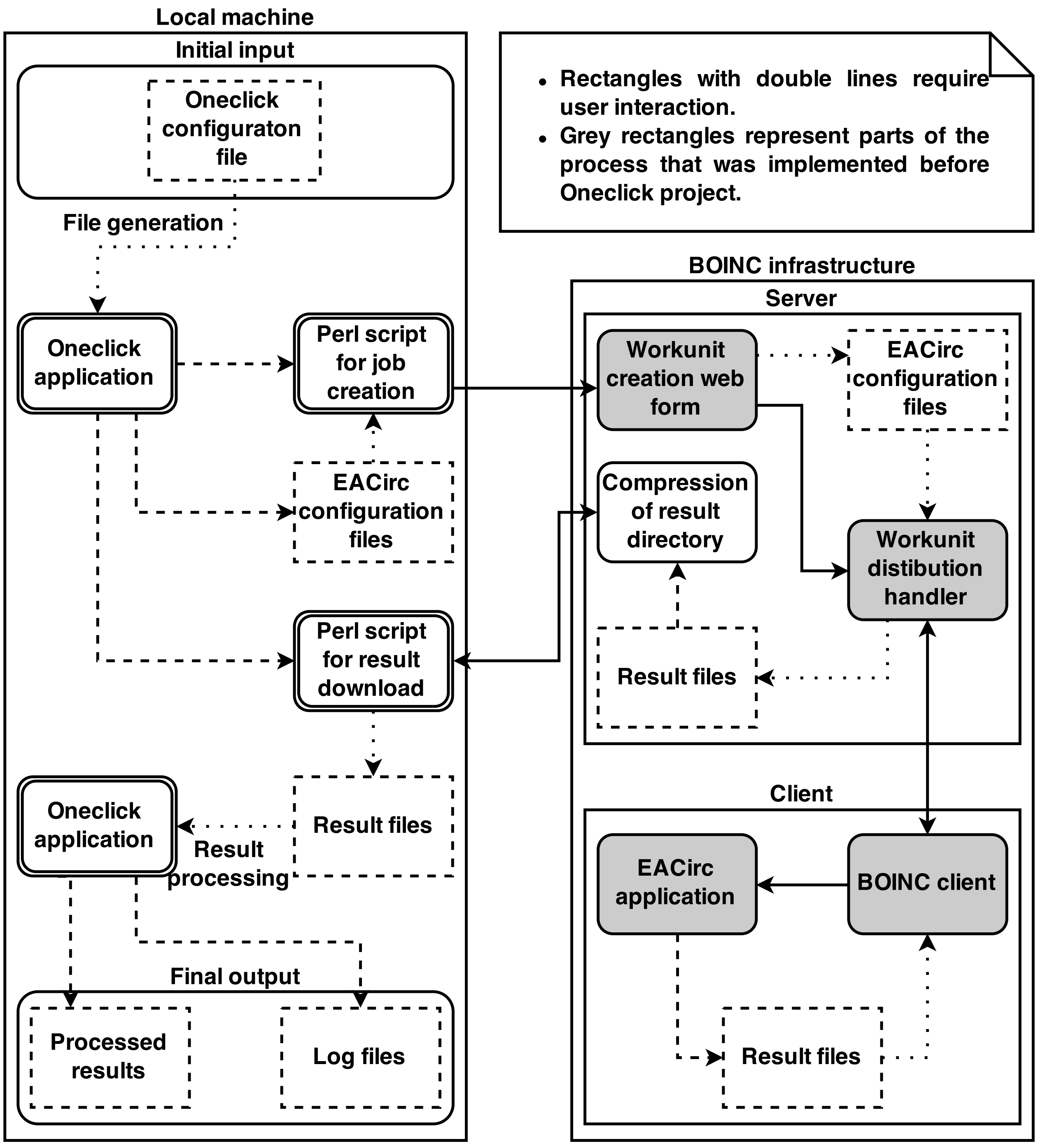
Local machine
Initial input
Oneclick configuraton file
File generation
Oneclick application
Perl script for job creation
EACirc configuration files
Perl script for result download
Result files
Oneclick application
Result processing
Final output
Processed results
Log files
• Rectangles with double lines require user interaction.
• Grey rectangles represent parts of the process that was implemented before Oneclick project.
BOINC infrastructure
Server
Workunit creation web form
EACirc configuration files
Compression of result directory
Workunit distibution handler
Result files
Client
EACirc application
BOINC client
Result files

Figure 2.1: Simplified work-flow of Oneclick tool.

# 3 Implementation of Oneclick

In this chapter, we will go through individual parts of automation process and describe their implementation and functionality.

## 3.1 Generation of files

In this part of process, scripts for upload and download of jobs to BOINC and configuration files for EACirc jobs are generated. User specifies what files will be created in configuration file for application. These files are required in next parts of the process.

Step is executed by application developed for this purpose. User creates single configuration file with instructions for application. Based on this file, scripts and configuration files are then created.

### 3.1.1 Oneclick options

Options for Oneclick application are set in it's configuration file. Configurations files for both EACirc and Oneclick are in `XML` format. Here is description of possible options that can be set in Oneclick configuration file.

**Algorithms**
This option sets for which algorithms will be created files.

**Rounds**
Application will create files for specified numbers of rounds with previously set algorithms.

**Specific rounds**
Some algorithms need different rounds for their proper working. In this option can be set range of rounds that will be used only with single specific algorithm. This option can be used multiple times.

**Additional settings**
This option is used for generating multiple configuration files with same algorithm and rounds, but another EACirc option different for each file. Range of values for any EACirc setting can be specified. This option can also be used multiple time.

**Workunit identifier**
When it's used, all generated files and jobs will have this as a suffix.

**Clones**
Option for setting number of jobs in single batch.

**BOINC project**
Specifies for which EACirc application on BOINC jobs will be generated. Currently, three instances of EACirc are running, one for main computations and two for testing new implementations.

**Static EACirc options**
These are options that remain unchanged for all generated configuration files. After discussion with CRoCS members and EACirc users we decided to have EACirc configuration file with general options set as part of the Oneclick file. This was reaction to a fact that even for large number of different batches, most of the options for EACirc have only one value.

```
<?xml version="1.0" ?>
<ONECLICK>
 <OC_OPTIONS>
  <!-- Files for algorithms from 1 to 5
       and rounds from 1 to 10 will be created -->
  <ALGORITHMS>1-5</ALGORITHMS>
  <ROUNDS>1-10</ROUNDS>
  <SPECIFIC_ROUNDS>
   <!-- Files for algorithm 6 with rounds
        from 11 to 15 will be created -->
   <ROUNDS algorithm="6">11-15</ROUNDS>
   <!-- Files for algorithm 7 with rounds 1,3,5,6,7 and 9
        will be created-->
   <ROUNDS algorithm="7">1 3 5-7 9</ROUNDS>
  </SPECIFIC_ROUNDS>
  <ADDITIONAL_SETTINGS>
   <!-- All files will have variants
        with 10000 and 20000 generations -->
   <VALUES path="EACIRC/MAIN/NUM_GENERATIONS">
    10000 20000
   </VALUES>
  </ADDITIONAL_SETTINGS>
  <!-- Names of created workunit will have suffix example -->
  <WU_IDENTIFIER>example</WU_IDENTIFIER>
  <!-- Each created batch will contain 100 jobs -->
  <CLONES>100</CLONES>
  <!-- Jobs will be created for main application -->
  <BOINC_PROJECT>EACirc</BOINC_PROJECT>
 </OC_OPTIONS>
 <EACIRC>
  <!-- Settings common for all jobs -->
 </EACIRC>
</ONECLICK>
```

Figure 3.1: Example configuration of Oneclick

From these options, only **Clones** and **BOINC project** options are mandatory. Additionally, file must contain **Static EACirc options**. Options **Algorithms**, **Rounds**, **Specific rounds** and **Additional settings** can be set to single value, set of values and range of values. See Figure 3.1 for example configuration.

### 3.1.2 Configuration files generation

After parsing of provided XML file, program will start generating configuration files and scripts. All settings from file are taken and for each combination of Algorithms, Rounds, Specific rounds and Additional settings a single EACirc configuration file is created. This process can take up to few minutes when creating large number of files e.g. 50000 or more.

### 3.1.3 Script files generation

Scripts are created from sample script files. Samples are written in Perl scripting language. In these samples, subroutines for creating single batch and downloading results from single batch are written. Subroutine call for each batch is added into scripts. Scripts handle all communication with server as well as user authentication on server.

## 3.2 Creation of jobs on BOINC server

During this step, generated configuration files for EACirc are uploaded to BOINC server and jobs for them are created.

### 3.2.1 Creation of single batch

In order to create single batch of jobs with same settings, BOINC web interface have to be used. User uploads single configuration file and enters settings for batch e.g. unique name or numbers of jobs in batch. Server then clones this job and creates multiple workunits.

### 3.2.2 Automated creation of multiple batches

This task is carried out by Perl upload script generated in first step. In the script there is subroutine that fill out form on server as if human would do it. For each batch, this process is repeated, so for each, there is subroutine call in generated script. Duration of this step depends on number of batches we are creating, where creation of single batch always takes from 3 to 5 seconds (independently on number of jobs in batch).

### 3.2.3 Authentication

Since BOINC server doesn't allow creation of jobs by unknown entities, user of this script has to be authenticated. At the beginning of the script, user is prompted for his BOINC name and password and this information is then used to log user in. Login details are stored only for the duration of the session.

## 3.3 Download of results from BOINC server

In this part, results are downloaded to local machine and stored for result processing.

### 3.3.1 Downloading results of single workunit

By default, server stores results in multiple directories. For each file, `4 byte` hash is calculated from it's name and file is stored in directory named after that value. As a result, files belonging to single job are stored in multiple directories. In order to access them, we have to calculate the hash for each file and download results one by one. This behavior can be overridden by implementing assimilator. Detailed description of our approach is described in Section 3.3.2.

For larger number of files, this process gets too lengthy to be viable for our purpose. Measurements showed that downloading only the most important results of computation of commonly used size would take several hours.

### 3.3.2 Assimilator

In BOINC environment, a code that will execute itself after completion of job is called assimilator. After discussion, we agreed that storing results of EACirc in separate directory on server would be beneficial. BOINC servers comes with sample assimilators and one of them moves results of all project jobs into single directory. We made changes to this code, so that files would be sorted into directories based on batch they belong to.

### 3.3.3 Downloading results of single batch

After implementation of assimilator, all files from single batch was stored in single directory. To avoid downloading of multiple small files we placed `PHP` script on server. This script creates zip archive of directory given in argument and puts this archive for download. Only one archive per batch is downloaded which reduced number of files downloaded as well as time needed for this step (see Section 3.3.4).

### 3.3.4 Comparison of download times

By using default ways for downloading of files we were able to download approximately 3 files per second. Since files were downloaded one by one, length of this step does not depend on our connection but rather on time needed to initialize download of single file, which is ineffective because initialization takes longer than download time of a small file.

When we used script for directory compression in combination with assimilator, `50` files per second were downloaded while speed of this process largely depends on our connection. Average file size was `20 KB`, same internet connection was used in both measurements.

### 3.3.5 Automated download of multiple batches

Perl script generated in the first part of the process uses `PHP` script on server. For each batch there are function calls that downloads and then extracts the archive on local machine. Files are then ready for post-processing.

## 3.4 Result processing

In this step, results downloaded from BOINC are post-processed and statistics for each batch are calculated. These statistics are then saved into human-readable result file where they are ready for further inspection.

### 3.4.1 Consistency and error checks

Before statistics calculation, we must check whether jobs in batch are in expected state.

Firstly, we have to check if all jobs in batch used same configuration file. In some cases, error on server occurs during workunit creation and one or more jobs in batch uses different config file for computation, rendering the jobs irrelevant in the context of the batch. Results of these jobs can't be taken into account when calculating statistics for the batch.

Secondly, we need to check log file of each job for errors and warnings. Results of job with errors are often corrupted or not present at all, therefore they can't be accounted into batch results. If only warnings are present in log, results of job can be still processed but user should be informed about warnings.

### 3.4.2 Processing of single batch

To obtain statistics of whole batch, we need to process results of all jobs. Format of results differs depending on evaluator used in EACirc computation.

**Categories Evaluator**
This evaluator is used in current computations. For each job, EACirc application determines whether given data were uniform or non-uniform based on their p-value calculated in the EACirc run. Result of batch is then ratio of uniform jobs and all jobs in batch.

**Top Bit Evaluator**
This evaluator was used in older computations. Success rate of job's distinguishers of non-random data is computed. In order to obtain results of batch, we need to calculate average of all success rates of jobs in batch.

### 3.4.3 Post-processors

Post-processor is part of Oneclick application that process batch files and calculates statistics of the batch. Since result processing is dependent on evaluator used in computation, we implemented multiple post-processors into application and application alone was designed so that adding new post-processors in the future would be possible with minimal effort. Post-processor type is specified by user at the beginning of result processing via command line argument. Two post-processors were implemented.

**p-values post-processor**
This post-processor is compatible with Categories Evaluator. Only logs from EACirc run are needed for post-processing. p-values are extracted from logs and stored in single file for each batch. Additionally, file with ratios of uniform jobs and all jobs for each batch is created. Post-processor is set as default in application and will be used if not specified otherwise by user.

**averages post-processor**
Is compatible with Top Bit Evaluator and was implemented for backward compatibility and recomputing of old results. Post-processor works only with log files from batch, success rates are extracted and stored in separate file for each batch. Averages of success rates of each batch are stored in single result file.

### 3.4.4 Processing of multiple batches

Oneclick application is able of checking consistency and errors and calculating results of multiple batches in one run.

At the beginning, consistency of batch is checked. In case this check fails, the batch is no further processed and this event is logged. If configuration files are consistent, logs are checked for errors and warnings. Jobs with errors are ignored in results calculating and logged.

Batch is then processed by specified post-processor and results are calculated and stored into output files. This process is repeated for each batch given for post-processing.

# 4 Verification of previous computations

In this chapter we will describe how was Oneclick used in recomputation of experiments performed as part of Ukrop's bachelor thesis [Ukr13]. Experiments examined EACirc's ability to distinguish random data from output of chosen candidate functions of projects eStream and SHA3.

We decided to rerun these computations in order to test actual usability of Oneclick tool along with implementation of EACirc that has been changing over the time. Due to time consuming manual testing of implemented EACirc projects and algorithms, new changes in code were tested only partially. With this process automated, we can compare our results to results of old experiments and detect any significant changes in application's behavior.

## 4.1 EACirc settings and output

For the sake of results comparability, our experiments had settings as close as possible to those used in previous experiments. We couldn't use exact same options because of changes in EACirc core and its configuration file.

Application in it's current state doesn't support arithmetic functions in circuit nodes, therefore functions `SUM`, `SUBS`, `ADD`, `MULT` and `DIV` weren't used in our experiments. Other bit-manipulating functions (`OR`, `AND`, `XOR`, `NOR`, `NAND`, `ROTL`, `ROTR` and `BITSELECTOR`) remained unchanged. We also decided to clone each job `100` times instead of `30` as in previous experiments. This change affect accuracy of obtained results not results alone.

Rest of the settings was taken from original experiments and was set to same or equivalent values.

### 4.1.1 Result format

Form of the results depends on used evaluator (see Section 3.4.2). In our experiments, we will be using Top Bit Evaluator.

Thorough the evolution, individuals in population distinguish random test vectors and test vectors that are outputs of specific algorithm. Test vectors change every `100`th generation and success rates of their guesses are evaluated in generation after test vector change. Success rate is represented as ratio of correctly guessed test vectors and all test vectors. From these success rates is then chosen maximum success rate. At the end of evolution, average of maximum success rates is calculated. This is result value of a single job. Result of batch is average of averages of maximum success rates (referred to as `AAM`).

Typically there are `20` individuals (distinguishers) in population, `1000` test vectors in a set, evolution runs for `30000` generations and there are `100` jobs in batch.

## 4.2 Random data distinguishing

Prior to running actual experiments, we need to examine results of EACirc application when distinguishing two truly random streams of data. We will test random data with same settings as will be used in the rest of experiments, but with variable number of individuals in population and test vectors in set.

Same experiments on random data were performed in Ukrop's thesis [Ukr13]. Due to changes in implementation we don't expect our values to be the same, but they should have following attributes in common:

- `AAM` value should be higher in experiments with more individuals in population because it's based on maximum success rates of individuals, therefore in bigger populations we have bigger chance of obtaining higher value.
- Experiments with higher number of test vectors in set should have lower `AAM`, since high number of vectors decreases chance of correct random guessing.

| | | Size of test vector set | | | | | |
|---|---|---|---|---|---|---|---|
| | | 200 | 500 | 1000 | 2000 | 5000 | 10000 |
| Population size | 5 | - | - | .5014 | - | - | - |
| | 10 | - | - | .5027 | - | - | - |
| | 20 | .5092 | .5056 | .5042 | .5029 | .5018 | .5013 |
| | 50 | - | - | .5060 | - | - | - |
| | 100 | - | - | .5078 | - | - | - |

Table 4.1: `AAM` values on random data

From the Table 4.1 we can see that `AAM` values indeed follow our presumptions as their value rises with individuals in population and decreases with number of test vectors in set. In further experiments, we will have `20` distinguishers in population and `1000` vectors in set, therefore our referential value is `0.5042`.[1] If an experiment result yield value roundable to `0.504`, we can rule data stream that experiment was testing as random.

1. Former experiments had referential value `0.52`

## 4.3 Oneclick settings

In this section we will present settings that were used for Oneclick application configuration for eStream and SHA3 projects respectively.

### 4.3.1 eStream project settings

We created experiments for previously tested combinations of algorithms and rounds. Each combination was run in three separate modes of cipher initialization:

- Cipher is initialized at the beginning of computation and all test vectors are generated using fixed key.
- All test vectors in single set were generated using fixed key, but key changed at every test vector set change.
- Every test vector was generated using different key.

Scripts and files generated with configuration file shown in Figure 4.1 were ready to use and no further changes were needed.

```
<?xml version="1.0" ?>
<ONECLICK>
 <OC_OPTIONS>
  <SPECIFIC_ROUNDS>
   <ROUNDS algorithm="4">1-8</ROUNDS>      <!-- Decim -->
   <ROUNDS algorithm="9">1 4</ROUNDS>      <!-- FUBUKI -->
   <ROUNDS algorithm="10">1-3 13</ROUNDS> <!-- Grain -->
   <ROUNDS algorithm="12">1 10</ROUNDS>    <!-- HERMES -->
   <ROUNDS algorithm="13">1-4 10</ROUNDS> <!-- LEX -->
   <ROUNDS algorithm="20">1-3 12</ROUNDS> <!-- Salsa20 -->
   <ROUNDS algorithm="25">1-13 32</ROUNDS><!-- TSC-4 -->
  </SPECIFIC_ROUNDS>
  <ADDITIONAL_SETTINGS>
   <!-- Setting for three modes of cipher initialization -->
   <VALUES path="EACIRC/ESTREAM/CIPHER_INIT_FREQ">0-2</VALUES>
  </ADDITIONAL_SETTINGS>
  <!-- 100 jobs in single batch -->
  <CLONES>100</CLONES>
  <!-- Main EACirc application -->
  <BOINC_PROJECT>EACirc</BOINC_PROJECT>
 </OC_OPTIONS>
 <EACIRC>
  <!-- Settings common for all jobs -->
 </EACIRC>
</ONECLICK>
```

Figure 4.1: Configuration of Oneclick for eStream project

### 4.3.2 SHA3 project settings

We again created experiments for previously tested combinations of algorithms and rounds.

Scripts and files generated by the configuration file shown in Figure 4.2 was ready to use and didn't need further changes.

```
<?xml version="1.0" ?>
<ONECLICK>
 <OC_OPTIONS>
  <SPECIFIC_ROUNDS>
   <ROUNDS algorithm="2">0-4</ROUNDS>       <!--ARIRANG-->
   <ROUNDS algorithm="3">0-4 17</ROUNDS>    <!--Aurora-->
   <ROUNDS algorithm="4">0-2 14</ROUNDS>    <!--Blake-->
   <ROUNDS algorithm="8">0-6 16</ROUNDS>    <!--Cheetah-->
   <ROUNDS algorithm="11">0-2 8</ROUNDS>    <!--CudeHash-->
   <ROUNDS algorithm="12">0-2 4</ROUNDS>    <!--DCH-->
   <ROUNDS algorithm="13">0-8 16</ROUNDS>   <!--Dynamic SHA-->
   <ROUNDS algorithm="14">1-13 17</ROUNDS>  <!--Dynamic SHA2-->
   <ROUNDS algorithm="15">1-3 8</ROUNDS>    <!--ECHO-->
   <ROUNDS algorithm="21">0-4 10</ROUNDS>   <!--Grostl-->
   <ROUNDS algorithm="22">0-3</ROUNDS>      <!--Hamsi-->
   <ROUNDS algorithm="23">0-7 42</ROUNDS>   <!--JH-->
   <ROUNDS algorithm="27">0-4 32</ROUNDS>   <!--Lesamnta-->
   <ROUNDS algorithm="28">0-8</ROUNDS>      <!--Luffa-->
   <ROUNDS algorithm="31">0-10 104</ROUNDS><!--MD6-->
   <ROUNDS algorithm="39">0-2 4</ROUNDS>    <!--SIMD-->
   <ROUNDS algorithm="44">
    0-5 10-24 80</ROUNDS>                   <!--Tangle-->
   <ROUNDS algorithm="46">0-9</ROUNDS>      <!--Twister-->
  </SPECIFIC_ROUNDS>
  <!-- 100 jobs in single batch -->
  <CLONES>100</CLONES>
  <!-- Main EACirc application -->
  <BOINC_PROJECT>EACirc</BOINC_PROJECT>
 </OC_OPTIONS>
 <EACIRC>
  <!-- Settings common for all jobs -->
 </EACIRC>
</ONECLICK>
```

Figure 4.2: Configuration of Oneclick for SHA3 project

## 4.4 Oneclick performance

In Table 4.2 we can see that all Oneclick operations took around 45 minutes. Our interaction was needed only when we was starting an operation (executing script or application). Creation and processing of same amount of jobs manually took around 12 hours of human work.[2]

| | SHA3 (14100 jobs) | eStream (11700 jobs) |
|---|---|---|
| File generation | 2 seconds | 2 seconds |
| Job creation | 7 minutes | 5 minutes |
| Results download | 6 minutes | 6 minutes |
| Post-procesing | 10 minutes | 5 minutes |
| Total | 23 minutes | 18 minutes |

Table 4.2: Duration of Oneclick operations

Each job we created, needed approximately 30 minutes to compute. All of our experiments took over 12900 hours (538 days) of processor time. Experiments were computed on 20 machines simultaneously for almost 27 days.

## 4.5 Comparison of results of experiments

In this section we will compare former results and results obtained from our experiments. Results of former experiment was taken from Ukrop's thesis [Ukr13]. Parentheses enclosing value indicate that EACirc wasn't able to detect any non-random traits in given data stream.

### 4.5.1 eStream project results

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | 1 | .988949 | .85 | .772914 | 1 | .954274 |
| 2 | .54 | .530987 | .54 | .530297 | (.52) | (.503635) |
| 3 | .53 | .514760 | .53 | .513972 | (.52) | (.503690) |
| 4 | (.52) | (.504505) | (.52) | (.504235) | (.52) | (.503749) |
| 5 | (.52) | (.504458) | (.52) | (.503833) | (.52) | (.503752) |
| 6 | (.52) | (.503845) | (.52) | (.503804) | (.52) | (.503736) |
| 7 | (.52) | (.503908) | (.52) | (.503745) | (.52) | (.503704) |
| 8 | (.52) | (.503925) | (.52) | (.503918) | (.52) | (.503859) |

Table 4.3: DECIM

2. According to Martin Ukrop, whose experiments are used for comparison.

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | (.52) | (.503656) | (.52) | (.503795) | (.52) | (.503642) |
| 4 | (.52) | (.503793) | (.52) | (.503738) | (.52) | (.503718) |

Table 4.4: Fubuki

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | 1 | .957750 | .67 | .521035 | (.52) | (.503871) |
| 2 | 1 | .933994 | .66 | .518083 | (.52) | (.503595) |
| 3 | (.52) | (.503808) | (.52) | (.503820) | (.52) | (.503778) |
| 13 | (.52) | (.503826) | (.52) | (.503907) | (.52) | (.503710) |

Table 4.5: Grain

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | (.52) | (.503992) | (.52) | (.503767) | (.52) | (.503770) |
| 10 | (.52) | (.503698) | (.52) | (.503795) | (.52) | (.503758) |

Table 4.6: Hermes

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | 1 | .985313 | 1 | .979921 | 1 | .986108 |
| 2 | 1 | .978462 | 1 | .979552 | 1 | .978720 |
| 3 | 1 | .962568 | 1 | .961080 | 1 | .962609 |
| 4 | (.52) | (.503924) | (.52) | (.503803) | (.52) | (.503714) |
| 10 | (.52) | (.503775) | (.52) | (.503979) | (.52) | (.503845) |

Table 4.7: LEX

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | .87 | .743646 | .67 | .509698 | (.52) | (.503734) |
| 2 | .87 | .749229 | .67 | .509256 | (.52) | (.503830) |
| 3 | (.52) | (.503797) | (.52) | (.503812) | (.52) | (.503794) |
| 12 | (.52) | (.503707) | (.52) | (.503946) | (.52) | (.503815) |

Table 4.8: Salsa20

| rounds | Cipher initialization mode | | | | | |
|---|---|---|---|---|---|---|
| | Once per run | | Once per set | | Once per vector | |
| | Former | Present | Former | Present | Former | Present |
| 1 | 1 | .991489 | 1 | .990565 | 1 | .990667 |
| 2 | 1 | .990171 | 1 | .990729 | 1 | .990315 |
| 3 | 1 | .990374 | 1 | .989744 | 1 | .990472 |
| 4 | 1 | .990667 | 1 | .991221 | 1 | .990582 |
| 5 | 1 | .990339 | 1 | .990587 | 1 | .990681 |
| 6 | 1 | .990388 | 1 | .990230 | 1 | .990112 |
| 7 | 1 | .990397 | 1 | .991198 | 1 | .990483 |
| 8 | 1 | .990640 | 1 | .990484 | 1 | .990531 |
| 9 | 1 | .986265 | 1 | .986687 | 1 | .985941 |
| 10 | 1 | .954197 | 1 | .954887 | 1 | .953543 |
| 11 | (.52) | (.503854) | (.52) | (.503898) | (.52) | (.504011) |
| 12 | (.52) | (.503850) | (.52) | (.503889) | (.52) | (.503804) |
| 13 | (.52) | (.503973) | (.52) | (.503751) | (.52) | (.503861) |
| 32 | (.52) | (.503710) | (.52) | (.503903) | (.52) | (.503836) |

Table 4.9: TSC-4

### 4.5.2 SHA3 project results

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997899 |
| 1 | 1 | .997988 |
| 2 | 1 | .997931 |
| 3 | 1 | .997976 |
| 4 | (.52) | (.504012) |

Table 4.10: ARIRANG

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .995859 |
| 1 | 1 | .779662 |
| 2 | 1 | .800408 |
| 3 | (.52) | (.504213) |
| 4 | (.52) | (.504059) |
| 17 | (.52) | (.504003) |

Table 4.11: Aurora

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .998008 |
| 1 | (.52) | (.503708) |
| 2 | (.52) | (.504165) |
| 14 | (.52) | (.504178) |

Table 4.12: Blake

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 1 | 1 | .791128 |
| 2 | (.52) | (.504037) |
| 3 | (.52) | (.504023) |
| 8 | (.52) | (.504081) |

Table 4.13: ECHO

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .998795 |
| 1 | 1 | .998100 |
| 2 | 1 | .998049 |
| 3 | 1 | .867146 |
| 4 | 1 | .866487 |
| 5 | (.52) | (.503891) |
| 6 | (.52) | (.504128) |
| 16 | (.52) | (.504012) |

Table 4.14: Cheetah

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .998955 |
| 1 | (.52) | (.503927) |
| 2 | (.52) | (.503999) |
| 8 | (.52) | (.503944) |

Table 4.15: CubeHash

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997757 |
| 1 | 1 | .996739 |
| 2 | 1 | .996757 |
| 3 | 1 | .954888 |
| 4 | .74 | .743968 |
| 5 | .61 | .613096 |
| 6 | .59 | .604849 |
| 7 | .59 | .611265 |
| 8 | (.52) | (.503998) |
| 16 | (.52) | (.504122) |

Table 4.16: Dynamic SHA

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .999027 |
| 1 | 1 | .769462 |
| 2 | (.52) | (.503943) |
| 4 | (.52) | (.504094) |

Table 4.17: DCH

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 1 | 1 | .946162 |
| 2 | .74 | .746828 |
| 3 | .75 | .749253 |
| 4 | .57 | .743968 |
| 5 | .60 | .613096 |
| 6 | .60 | .578779 |
| 7 | .61 | .611265 |
| 8 | .60 | .622003 |
| 9 | .61 | .621060 |
| 10 | .61 | .617805 |
| 11 | (.52) | (.504230) |
| 12 | (.52) | (.504153) |
| 13 | (.52) | (.504182) |
| 17 | (.52) | (.503998) |

Table 4.18: Dynamic SHA2

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .737774 |
| 1 | .58 | .715732 |
| 2 | .58 | .727071 |
| 3 | (.52) | .508260 |
| 4 | (.52) | .508280 |
| 10 | (.52) | (.504088) |

Table 4.19: Grøstl

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997456 |
| 1 | 1 | .992742 |
| 2 | 1 | .989812 |
| 3 | 1 | .989312 |
| 4 | 1 | .995962 |
| 5 | 1 | .997072 |
| 6 | 1 | .992859 |
| 7 | (.52) | (.503831) |
| 42 | (.52) | (.504088) |

Table 4.20: JH

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .987366 |
| 1 | (.52) | (.504014) |
| 2 | (.52) | (.504165) |
| 3 | (.52) | (.504172) |

Table 4.21: Hamsi

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .999114 |
| 1 | 1 | .997608 |
| 2 | 1 | .997693 |
| 3 | 1 | .997920 |
| 4 | 1 | .997618 |
| 5 | 1 | .982047 |
| 6 | .88 | .879778 |
| 7 | .65 | .637511 |
| 8 | .53 | .530127 |
| 9 | (.52) | (.504226) |
| 10 | (.52) | (.504132) |
| 104 | (.52) | (.504178) |

Table 4.22: MD6

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997553 |
| 1 | 1 | .997540 |
| 2 | 1 | .997885 |
| 3 | (.52) | (.504076) |
| 4 | (.52) | (.503970) |
| 32 | (.52) | (.504003) |

Table 4.23: Lesamnta

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997700 |
| 1 | 1 | .997768 |
| 2 | 1 | .997961 |
| 3 | 1 | .997629 |
| 4 | 1 | .997472 |
| 5 | 1 | .997770 |
| 6 | 1 | .997594 |
| 7 | (.52) | (.503949) |
| 8 | (.52) | (.503935) |
| 9 | (.52) | (.504077) |

Table 4.24: Twister

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997909 |
| 1 | 1 | .996927 |
| 2 | 1 | .994601 |
| 3 | 1 | .992327 |
| 4 | .75 | .746475 |
| 5 | .75 | .745629 |
| 6 | .74 | .743688 |
| 7 | .74 | .742861 |
| 8 | (.52) | (.504012) |

Table 4.25: Luffa

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .997892 |
| 1 | 1 | .991669 |
| 2 | 1 | .995297 |
| 3 | .85 | .888356 |
| 4 | .84 | .851132 |
| 5 | .80 | .840793 |
| 10 | .64 | .648893 |
| 11 | .63 | .648754 |
| 12 | .64 | .650617 |
| 13 | .64 | .649834 |
| 14 | .64 | .648932 |
| 15 | .64 | .650467 |
| 16 | .64 | .648025 |
| 17 | .60 | .622259 |
| 18 | .60 | .627164 |
| 19 | .60 | .590435 |
| 20 | .60 | .590683 |
| 21 | .54 | .559599 |
| 22 | .54 | .558740 |
| 23 | (.52) | (.504081) |
| 24 | (.52) | (.503924) |
| 80 | (.52) | (.504165) |

Table 4.26: Tangle

| rnds | Results | |
|---|---|---|
| | Former | Present |
| 0 | 1 | .991073 |
| 1 | (.52) | (.504102) |
| 2 | (.52) | (.503945) |
| 4 | (.52) | (.503958) |

Table 4.27: SIMD

## 4.6 Conclusions based on comparison

In Sections 4.5.1 and 4.5.2 we compared outputs of current implementation of EACirc to outputs of older implementation. Note that present results roundable to `0.504` are equivalent to former value `0.52` and present results higher than `0.9` are equivalent to former value `1`.[3]

Most of the time EACirc outputted values very close to former results and every time former EACirc detected non-randomness in stream, present EACirc detected it too.

Present EACirc found distinguishers with notably lower success rate in outputs of following algorithms: Grain (Table 4.5), Salsa20 (Table 4.8), Aurora (Table 4.11), ECHO (Table 4.13), Cheetah (Table 4.14) and DCH (Table 4.17).

Interestingly, present EACirc found non-random traits in product of Grøstl hash function (Table 4.19) weakened to `3` and `4` rounds even when former EACirc evaluated these outputs as completely random.

Reason behind these irregularities can be one of or combination of following reasons:

- Full range of node functions used in former computations isn't implemented in current version of software (see Section 4.1).
- Not completely reproducible settings for evolution (mutation, sexual cross-overs, probabilities for genetics, etc.).
- Other reason caused by core reimplementation.

Also note that we weren't using newly implemented Categories Evaluator that should provide better results. This evaluator wasn't used because we wouldn't be able to compare our results to the old ones if we did.

3. Former value `0.52` means that data are indistinguishable from random data whereas value `1` means that EACirc can successfully distinguish given data from random data with high probability. See Section 4.1.1 for details about EACirc output.

# 5 Alternative statistical test suites overview

In this chapter we will present alternative approaches to automated testing of data stream randomness. Further in the chapter we will show results of chosen suites on multiple pseudo-random number generators to compare their ability to spot non-randomness in data.

## 5.1 Statistical test suites

### 5.1.1 NIST Statistical Test Suite

*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* is battery of statistical tests implemented by National Institute of Standards and Technology [Ruk+10].

The suite is one of the most commonly used tools for testing statistical properties of output of arbitrary algorithm or pseudo-random generator but it's usable on any bit stream. Main aim of this battery is to detect non-randomness in given data. The randomness of data is determined according to the results of following statistical tests.

**Frequency Test (within a block)**
Calculate ratio of `1`s and `0`s in whole stream or block and checks whether it's close to `0.5`.

**Runs Test**
Sequences of same bits and same lengths are counted and this value is compared to value produced by truly random generator.

**Test For The Longest Run Of 1s In A Block**
Compares length of longest sequence of `1`s in block of certain length to predefined value.

**Random Binary Matrix Rank test**
Test is focused on rank of disjoint sub-matrices of entire data stream. Aim of this test is to check substrings for linear dependence in the sequence.

**Discrete Fourier Transform Test**
`1`s are transformed to `+1` and `0`s to `-1`. On this stream is then applied discrete Fourier transformation.

**(Non-)overlapping Template Matching Test**
Occurrences of certain target substrings are counted within the stream and compared to predefined values.

**Maurer's Universal Statistical Test**
Checks distances between same patterns.

**Linear Complexity Test**
Checks how long is stream until it begins to repeat itself.

**Serial Test**
Measures frequency of every `m`-bit sequence in stream and compares it to predefined frequencies.

**Approximate Entropy Test**
Counts every overlapping `m`-bit and `(m+1)`-bit sequence and then compares their respective frequencies to frequencies expected in truly random data stream.

**Cumulative Sum Test**
`0`s and `1`s are transformed to `-1` and `+1` respectively. Sums of blocks of various lengths are then calculated and compared to predefined values.

**Random Excursions Test**
`0`s and `1`s are transformed to `-1` and `+1` respectively. Cumulative sums states within a cycle are then counted are compared to values expected of random data.

### 5.1.2 DIEHARD

*Diehard Battery of Tests of Randomness* was developed at Florida State University by George Marsaglia. Original CD-ROM with test battery was published in 1995 [Mar95]. Suite implements following tests to determine quality of a pseudo-random generator.

**Birthday spacing**
Random points on certain interval are generated. Spacings between points should be distributed in a certain way.

**Overlapping permutations**
Sequences of five numbers are generated. All possible orders (`5!`) should occur with equal probability.

**Ranks of matrices**
Same test as Matrix Rank test in NIST STS (see Section 5.1.1).

**Monkey test**
Some sequences of generated bits are treated as words. Missing words are then counted and compared to value expected of truly random stream.

**Count the 1s**
`1`s are counted on certain positions or in sequences. The counts are then converted into words and number of five letter words is checked.

**Parking lot test**
Random positions for unit circles are generated. Circles are then placed into `100x100` square. Should new circle overlap with another, this one is skipped. Number of successfully placed circles is counted after `12000` tries.

**Minimum distance test**
8000 random points positions are generated and then placed into $10^4$x$10^4$ square. Minimal distance between two points is then found and checked.

**Random spheres test**
4000 random points within cube of edge 1000 are generated. Each point is then center of sphere whose radius is distance to the closest point. Volume of smallest sphere is checked.

**The squeeze test**
Random floats from interval (0,1) are generated and multiplied with $2^{31}$ until 1 is reached. Number of floats needed should be close to predefined value. This process is repeated 100000 times.

**Overlapping sums test**
Sums of 100 consecutive floats from interval (0,1) generated by random generator should follow certain distribution.

**Runs test**
Long sequence of floats from interval (0,1) is generated. Ascending and descending sequences are counted and compared to expected values.

**The craps test**
2000000 rounds of game Craps are played. Number of throws per game should be close to expected value.

### 5.1.3 DIEHARDER

This testing suite was developed by Robert G. Brown at the Duke University. It's main aim is to make process of testing randomness of bit stream easy while it is still possible for researchers to control tests on low level.

Suite is not just descendant of Diehard, although it mainly includes tests from this suite in enhanced way. Tests from NIST STS are being incorporated into battery as well as entirely new tests developed by it's authors and other users. This tool features many improvements over Diehard i.e. full extensibility, simple user interface or full open surce access [Bro04].

### 5.1.4 TestU01

*TestuU01* is a library for empirical testing of random number generators implemented in ANSI C language. It was developed at Université de Montréal mainly by Pierre L'Ecuyer [LS07].

The library implements statistical tests present mainly in NIST STS or in Dieharder. Detailed description of battery's tests can be found in documentation [LEc09]. TestU01 also implements wide variety of pseudo-random number generators (further referred to as PRNG) from many different categories.

Statistical suite implements various batteries intended for different purposes and including different set of tests. Following batteries are present:

**SmallCrush Battery**
Set of tests intended to make evaluation of a PRNG quickly.

**Crush Battery**
Intended to determine goodness of a PRNG with high certainty.

**BigCrush Battery**
Heavyweight collection of tests for thorough examination of a generator.

**Rabbit Battery**
Tests was selected with aim to detect wide range of non-randomness as well as to run quickly.

**Aplhabit Battery**
Tests aiming to examine hardware random number generators.

**BlockAlphabit Battery**
Alphabit Battery applied to data after reordering the stream by blocks of different size.

**PseudoDiehard Battery**
Same set of tests that Diehard test suite implements.

**FIPS_140_2**
Set of tests from NIST standard FIPS PUB 140-2 [Eva+01].

### 5.1.5 PractRand

*PractRand* is a C++ library including pseudo-random generators as well as statistical tests for PRNGs, developed by Chris Doty-Humphrey [Dot10]. It provides convenient user interface not just for research but also for practical use and variety of random generator algorithms from multiple categories.

Statistical suite implements standard tests from Diehard along with more original tests created by author. Additionally, battery is not limited in length of stream it can process, provides preliminary results to the tests without need to restart the testing and supports multithreaded testing for higher performance on multiple CPU cores.[1]

### 5.1.6 RaBiGeTe

*RaBiGeTe* is a randomness testing framework developed for Windows and including user friendly GUI as well as multi-thread support. Input stream

1. Author states that PractRand was tested on 100 terabytes long sequence and is expected to be able to process streams as long as few exabytes

can be supplied to program via binary file or via DLL. Library implements 24 statistical tests, among them are improved versions of NIST's STS tests, chosen statistical tests described by Knuth [Knu97] and author's original tests. Detailed description of implementation and interface can be found in documentation [Pi04].

### 5.1.7 CryptoStat

*CryptoStat Library* is Java framework for statistical analysis of block ciphers and MACs [KS14]. It uses Bayesian methodology to test of cryptographic function's output. Suite implements following tests:

**The linear approximation test**
Checks whether results of adding different bit groups on plaintext and ciphertext `mod 2` are uniformly distributed.

**The coincidence test**
Examine bit groups on ciphertext. Checks whether fixed target value occurs in certain `g`-bit group with probability $2^{-g}$.

**The input-output independence test**
Tests independence of certain bit group on plaintext on different bit group on ciphertext.

**The complement test**
Examined are two ciphertexts that are products of encryption of two plaintexts that are complementing each other with two keys that are complementing each other. Checks whether the same bit groups on the ciphertexts are independent.

**The cipher independence test**
Tests bit groups on one ciphertext. Certain bit group should be independent of another bit group.

**The strong avalanche test**
Compares unaltered ciphertext and ciphertext that was product of plaintext and key with each bit flipped. Differences between bit groups on ciphertexts should be uniformly distributed.

**The uniformity test**
Checks uniform distribution in bit groups on single ciphertext.

Tested cryptographic primitive must implement CryptoStat's interface for functions [Kam+14]. Currently implemented functions are PRESENT (`80`-bit key), Kasumi, ThreeWay, Blowfish, AES (`128`-bit key), IDEA, SipHash and SQUASH.

All tests use generated ciphertext samples by key-plaintext pairs. Number of these pairs is specified by parameter `S`. Then, for each test run `2*S+2` encryption keys are created: all-0s key, all-1s key, S random keys and S consecutive keys starting from random value. For all combinations of keys and plaintexts are generated ciphertexts resulting in $(2*S+2)^2$ unique strings. Furthermore, every test can be redone T times.

### 5.1.8 YAARX

*Yet Another Toolkit for Analysis of ARX Cryptographic Algorithms* is a set of programs for differential analysis of cryptographic algorithms belonging to ARX family, developed at University of Luxembourg [BV14].

ARX represents class of symmetric-key algorithms that encrypts data using simple operations such as addition, rotation, XOR, bit rotations or bit shifting. Such algorithms are for example the block ciphers FEAL, RC5, TEA, the stream ciphers Salsa20 or the hash functions Skein, BLAKE or SipHash.

Toolkit is implemented for ciphers TEA and XTEA and provides means to be applicable to many ARX algorithms. Additionally this tool was used for analysis of algorithms SPECK and RAIDEN.

Implementation of YAARX provides methods for the calculation of the differential probabilities of ARX bit operations such as XOR, modular addition, multiplication, bit shift or bit rotation as well as larger structures built from them. Fully automated way to look for high-probability differential trails in ARX algorithms is also provided.

## 5.2 Comparison of chosen test suites

In this section, we will present results of three test suites applied to multiple pseudo-random number generators. Namely we will compare TestU01 (Section 5.1.4), PractRand (Section 5.1.5) and RaBiGeTe (Section 5.1.6) on outputs of algorithms HC-256, Mersenne Twister, RC4 and two weakened Salsa20 variants.

The first three algorithms was chosen because of their wide usage and quality. Salsa20 variants were chosen because we expect them to contain non-random traits and we want to see whether any suites will rule out these data streams as non-random.

Note, that another comparisons of test suites were conducted in the past as well as testing of certain suite on wide range of functions. EACirc was compared to NIST STS and Dieharder in Ukrop's bachelor thesis [Ukr13] and in following SeCrypt 2013 paper [ŠUM13]. Dieharder and TestU01 were also compared [Jak14] and TestU01 alone was tested on various PRNGs [LS07].

### 5.2.1 Comparison results

| | TestU01 | PractRand | RaBiGeTe |
|---|---|---|---|
| HC-256 | PASS | PASS | PASS |
| Mersenne Twister | 0/2/2 | 256GB | PASS |
| RC4 | PASS | 1TB | PASS |
| Salsa20 (3 rounds) | 13/-/- | 4KB | 2KB |
| Salsa20 (4 rounds) | PASS | 16GB | 32MB |

Table 5.1: Results of test suites on multiple algorithms. Data were taken from PractRand documentation page [Dot10].

PASS means that no bias in data was found by any of the tests in battery. Any other value in columns with PractRand and RaBiGeTe shows how long stream had to be for battery to spot non-randomness. Values in TestU01 columns shows number of failed tests in SmallCrush/Crush/BigCrush batteries respectively. Dash means that tests in previous battery failed too many times, stream was no further examined and ruled as non-random.

As we can see in Table 5.1, out presumptions about quality of HC-256 generator were fulfilled as all tests passed.

Next two generators didn't pass PractRand battery, but to spot non-randomness was used nontrivial amount of data. Mersenne Twister also failed small number of tests in TestU01 batteries.

Salsa20 weakened to 3 rounds also fulfilled our presumption, as it was ruled as non-random by all three batteries after small amount of data and tests. Salsa20 with 4 rounds failed PractRand and RaBiGeTe but passed all TestU01 batteries, which could mean that TestU01 is less sensitive or doesn't have right set of tools to spot non-random traits in this data stream.

From presented results, we can conclude that in these cases, PractRand is more sensitive to non-randomness at cost of large amount of data processed.

# 6 Conclusions

In this work we introduced a tool for management and processing of EACirc experiments called Oneclick. Oneclick consists of multiple smaller scripts and applications for which we provided documentation as part of source code, implementation overview in Chapter 3 and user guide with instructions for building and comfortable use as part of EACirc GitHub wiki [Š+12]. Application is multiplatform, easily modifiable and extensible and dependent on EACirc only in parts that are absolutely necessary.

In order to test functionality of introduced tool, we fully reproduced EACirc results from Martin Ukrop's bachelor thesis [Ukr13]. When using complete implementation of Oneclick we were able to reproduce these experiments with minimal user interaction and in fraction of time that was needed to perform older experiments. During testing we experienced multiple drawbacks mostly when downloading results from BOINC server. To overcome them we had to modify default behavior of server (see Section 3.3 for details) allowing us to download results very efficiently.

We compared results of our experiments to results of older experiments to see whether newer implementation of EACirc has better, same or worse ability to detect non-randomness in data stream than older implementation. Based on comparison in Chapter 4 we concluded that EACirc preserved its distinguishing qualities. In multiple cases we obtained better or worse results than we expected. This could be caused by different set of functions used in circuit nodes or changes in implementation. Our experiments were done using evaluator that is no longer in use, therefore it is possible that our following computation will deliver better success in distinguishing outputs of cryptographic functions.

In Chapter 5 we prepared overview of used statistical test suites for testing data for occurrence of non-random traits. At the end of chapter we present comparison of performances of few chosen suites on multiple algorithms.

In the future, we plan to recompute experiments again but with use of current evaluator, settings and circuit functions. After these tests we will have better knowledge about EACirc's abilities. Given our current capability to run many experiments without effort we can try more combinations of settings in shorter time. It is also possible that we pick up idea to create alternative Oneclick as server service, thus removing need for workunit creation from local machine and local result processing when user doesn't need to modify the process. Additionally, we can extend Oneclick from EACirc to wider range of projects running on our BOINC server.

# Bibliography

[Bro04] R. G. Brown. (2004). Dieharder, A Random Number Test Suite. version 3.31.1, Duke University Physics Department, [Online]. Available: http://www.phy.duke.edu/~rgb/General/dieharder.php (visited on 04/21/2015).

[BV14] A. Biryukov and V. Velichkov, "Automatic Search for Differential Trails in ARX Ciphers", 2014. [Online]. Available: https://eprint.iacr.org/2013/853.pdf (visited on 04/16/2015).

[Cen] Centre for Research on Cryptography and Security. [Online]. Available: http://www.fi.muni.cz/crocs (visited on 05/06/2015).

[Dot10] C. Doty-Humphrey. (2010). PractRand. version 0.92, [Online]. Available: http://pracrand.sourceforge.net (visited on 04/29/2015).

[Eur05] European Network of Excellence for Cryptology. (2005). eStream project, Call for stream cipher primitives, [Online]. Available: http://www.ecrypt.eu.org/stream/call/ (visited on 05/06/2015).

[Eva+01] D. L. Evans *et al.*, "Security requirements for cryptographic modules", FIPS PUB 140-2, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf (visited on 04/29/2015).

[Jak14] K. S. Jakobsson, "Theory, Methods and Tools for Statistical Testing of Pseudo and Quantum Random Number Generators", 2014. [Online]. Available: http://liu.diva-portal.org/smash/get/diva2:740158/FULLTEXT01.pdf (visited on 04/29/2015).

[Kam+14] A. Kaminsky *et al.* (2014). The CryptoStat Library, Department of Computer Science Rochester Institute of Technology, [Online]. Available: http://www.cs.rit.edu/~ark/parallelcrypto/cryptostat/doc/ (visited on 04/16/2015).

[Knu97] D. E. Knuth, *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997, ISBN: 0201896842.

[KS14] A. Kaminsky and J. Sorrell, "CryptoStat, A Bayesian Statistical Testing Framework for Block Ciphers and MACs", 2014. [Online]. Available: http://www.cs.rit.edu/~ark/students/jls6190/report.pdf (visited on 04/16/2015).

[LEc09] P. L'Ecuyer. (2009). TestU01. version 1.2.3, Université de Montréal, [Online]. Available: http://simul.iro.umontreal.ca/testu01/tu01.html (visited on 04/29/2015).

[LS07] P. L'Ecuyer and R. Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators", in *ACM Transactions on Mathematical Software*, vol. 33, 4, article 22, 2007.

[Mar95] G. Marsaglia. (1995). Diehard Battery of Tests of Randomness, Florida State University, [Online]. Available: http://www.stat.fsu.edu/pub/diehard/ (visited on 04/21/2013).

[Nat07] National Institute for Standards and Technology. (2007). SHA-3, Cryptographic hash algorithm competition, [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html (visited on 05/06/2015).

[Pi04] C. Pi. (2004). RaBiGeTe Documentation. version 2.0.0, [Online]. Available: http://cristianopi.altervista.org/RaBiGeTe_MT/ (visited on 04/29/2015).

[Ruk+10] A. Rukhin *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", 2010. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf (visited on 04/21/2015).

[Š+12] P. Švenda, M. Ukrop, M. Prišťák, *et al.* (2012). Eacirc, Framework for autmatic search for problem solving circuit via evolutionary algorithms, Laboratory of Security and Applied Cryptography, Masaryk University, [Online]. Available: http://github.com/petrs/EACirc (visited on 05/06/2015).

[Str02] F. Streichert, "Introduction to Evolutionary Algorithms", pp. 4–6, 2002. [Online]. Available: http://www.ra.cs.uni-tuebingen.de/mitarb/streiche/publications/Introduction_to_Evolutionary_Algorithms.pdf (visited on 04/16/2015).

[ŠUM13] P. Švenda, M. Ukrop, and V. Matyáš, “Towards cryptographic function distinguishers with evolutionary circuits”, 2013. [Online]. Available: http://www.fi.muni.cz/~xsvenda/papers/secrypt2013/Svenda_EACirc_SeCrypt2013.pdf (visited on 04/29/2015).

[Ukr13] M. Ukrop, “Usage of evolvable circuit for statistical testing of randomness”, bachelor thesis, Faculty of Informatics Masaryk University, 2013. [Online]. Available: http://is.muni.cz/th/374297/fi_b/ (visited on 04/16/2015).

[Uni] University of California, Berkeley. [Online]. Available: http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing (visited on 05/07/2015).

# A Data attachment

The data attachment contains source codes and results of our experiments sorted in following structure:

**`eacirc_oneclick`**
Source code of EACirc and Oneclick on GitHub. Oneclick related code is in directory `oneclick` (contains copy of entire `oneclick` branch of EACirc's repository with last commit `c8fb964` from 2015-05-14).

**`eacirc_wiki`**
Wiki of EACirc project on GitHub. Contains user documentation for Oneclick (contains copy of wiki repository with last commit `cd4758e` from 2015-05-14).

**`results_estream`**
Raw results of experiments distinguishing outputs of eStream algorithms from random data (used in Section 4.5.1).

**`results_sha3`**
Raw results of experiments distinguishing outputs of SHA-3 algorithms from random data (used in Section 4.5.2).

**`results_random`**
Raw results of experiments distinguishing random data from random data (used in Section 4.2).

**`thesis_src`**
Source files of thesis, including figures, images and bibliography used in text.