УДК 621.391:519.72

Казакова Н.Ф., Щербина Ю.В.
(Одесский государственный экономический університет)

ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ СТАТИСТИЧЕСКОГО ТЕСТА МАУРЕРА ДЛЯ АНАЛИЗА КРИПТОГРАФИЧЕСКИХ ГЕНЕРАТОРОВ ПСЕВДОСЛУЧАЙНЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Постановка проблемы в общем виде и ее связь с научными и практическими задачами. В настоящее время интерес к потоковым шифрам объясняется совокупностью задач, стоящих перед современной криптографией. К их числу относят: обеспечение долговременной стойкости, создание криптографических схем с низкой ресурсоемкостью (*low footprint*) и небольшим потреблением энергии, обеспечение высокой производительности и разработку легко переносимых алгоритмов (*algorithm agility*). Общей особенностью перечисленных проблем является стремление обеспечить высокую производительность и качество шифрования при одновременной экономии временных и вычислительных ресурсов. Именно способность потоковых шифров осуществлять побитовое преобразование и делает их весьма пригодными для реализации данных целей, а это, в свою очередь, заставляет специалистов, работающих в этой области сосредотачивать усилия на поиске новых эффективных решений.

Актуальность перечисленных задач, привела к тому, что европейским криптологическим сообществом ECRYPT был объявлен открытый конкурс (2004…2008г.г.) на разработку новых потоковых шифров – eSTREAM (*ECRYPT Stream Cipher Project*) [1] с целью выявления наиболее достойного соискателя на использование в качестве стандарта для стран европейского сообщества.

Потоковые шифры по сути своей пытаются имитировать концепцию одноразового гаммирующего блокнота, обоснованную в свое время Клодом Шенноном [2]. Используя короткий ключ для генерации шифрующей гаммы, потоковый алгоритм стремиться сформировать последовательность, которая в максимальной степени походила бы на случайную. Другими словами, сформированная псевдослучайная последовательность (ПСП) по своим статистическим свойствам должна содержать символы, подчиняющиеся равномерному закону распределения. При этом сразу же возникает проблема оценки того, насколько точно выполняется это требование. Проблема построения датчиков хороших псевдослучайных последовательностей существует и в других прикладных областях (например, в моделировании), однако требования к ним в криптографии несоизмеримо выше.

Для оценки степени «равномерности» распределения вероятностей символов в формируемых ПСП обычно используют различные наборы тестов, к числу которых, в первую очередь следует отнести пакет NIST STS [3]. Он включает набор из 16-ти тестов и методику их использования.

Кроме того, известны и другие пакеты тестов, созданные для нужд криптографии. К их числу относится набор статистических тестов под названием Diehard [4], предназначенный для определения качества последовательности случайных чисел. Эти тесты были разработаны Джорджем Марсальей (*George Marsaglia*). Он включает 12 тестов и доступен по адресу http://stat.fsu.edu/pub/diehard/.

По адресу http://www.isi.qut.edu.au/resources/cryptx/ можно связаться с разработчиками пакета тестов Crypt-X [5] и получить программное обеспечение и руководство по их применению.

Успешный результат испытаний проектируемого генератора с применением всего набора этих тестов дает основания надеяться на то, что формируемая генератором последовательность неотличима от «настоящей» случайной последовательности. Однако на практике полный пакет тестов применяется только в условиях комплексных испытаний

готового продукта. В процессе же поиска хорошей архитектуры генератора приходится постоянно прибегать к испытанию его «промежуточного» варианта или отдельных криптографических преобразований, составляющих алгоритм шифрования. Поэтому, разработчики, зачастую, сами разрабатывают тесты, дающие возможность контролировать качество создаваемого нового шифра.

Из анализа логики построения перечисленных тестовых пакетов вытекает, что в процессе проектирования для предварительной оценки генератора лучше всего применять тесты на основе критерия согласия χ^2 или частотный тест. Если эти тесты не выполняются, то, как указывается в рекомендациях к тестовым пакетам, дальнейшее тестирование формируемой генератором ПСП не имеет смысла. Если эти тесты успешно выполняются, то следующим хорошим тестом, позволяющим убедится в том, что разработчик находится на правильном пути, может быть рекомендован тест Маурера [2,6]. Это объясняется тем, что в отличие от иных тестов, у него есть два преимущества. Во-первых, он не ориентирован на обнаружение какого-либо конкретного дефекта, а позволяет обнаруживать широкий класс аномалий, могущих иметь место в составе ПСП, формируемых эргодическими стационарными источниками с конечной памятью. Во-вторых, этот тест позволяет, в случае если тестируемый генератор будет применяться в качестве источника ключей, получать взвешенную оценку величины потенциального ущерба наносимого безопасности системы.

Целью данной статьи является акцентирование внимания на особенностях программной реализации и применения универсального теста Маурера для обнаружения наличия дефектов в тестируемых криптографических последовательностях. Этот тест основан на идее Зива для универсальных алгоритмов кодирования о том, что случайную последовательность с равномерным законом распределения вероятностей нельзя сколько-нибудь заметно сжать без потери информации. Его тестовая статистика тесно связана с посимвольной энтропией потока, который, по мнению ее автора, является правильной мерой качества формируемой псевдослучайной последовательности.

В соответствии с идеей, лежащей в основе теста, двоичная n-битная последовательность

$\square = \square_1, \square_2, \ldots, \square_n$,

формируемая на выходе генератора ПСП, разделяется на L-битные непересекающиеся последовательные блоки, размер которых может изменяться от двух до шестнадцати символов. Величина L является параметром, определяемым в процессе тестирования. Он связан с величиной энтропии на символ источника, порождающего ПСП.

После разделения на L-битные блоки тестируемая последовательность делится на два сегмента: сегмент инициализации, включающий Q L-битных блоков и тестируемый сегмент, включающий K следующих за ними блоков такого же размера, так, как это показано на рис. 1.

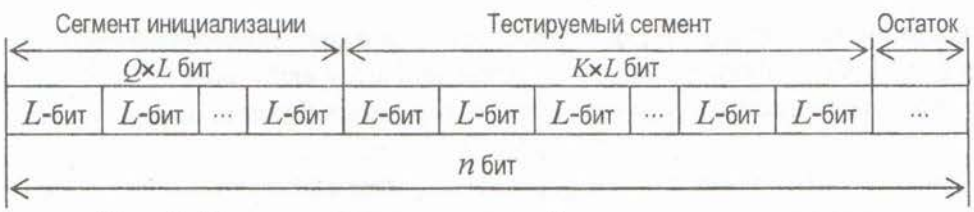

Рис. 1. Разделение тестируемой последовательности

Если длина последовательности n превышает суммарный размер сегментов инициализации и тестирования, то оставшиеся блоки просто отбрасываются и в тестировании не участвуют. Общая протяженность участка тестируемой последовательности при этом составляет $(Q + K)L$ бит.

Далее создается массив размером 2^L, в ячейках которого запоминаются номера последних позиций i, $1 \le i \le Q$, на которых были обнаружены L-разрядные блоки конкретного вида. До начала тестирования все они должны быть заполнены, то есть каждая

L-разрядная группа должна встретится на участке инициализации хотя бы один раз. По этой причине величина сегмента инициализации Q выбирается из условия $Q \geq 10 \cdot 2^L$.

Тестирующая функция f_n определяется как средний логарифм (по основанию 2) от интервалов между одноименными L-битными блоками $i - T_j$, где T_j – номер предыдущей позиции, на которой был обнаружен текущий i-й блок. Она имеет следующий вид

$$f_n = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \log_2 (i - T_j)$$

На основании этой функции, в соответствии с рекомендациями, изложенными в [2], вычисляется показатель

$$P_v = erfc\left(\left| \frac{f_n - m_L}{\sqrt{2\sigma}} \right| \right),$$

где $erfc$ – есть дополнительная функция ошибки, m_L – теоретически ожидаемое выборочное среднее значение, вычисляемое из статистики для данного L-битного блока. Теоретическое значение стандартного отклонения определяется как

$$\sigma = c \sqrt{\frac{D_L}{K}},$$

где

$$c = 0.7 - \frac{0.8}{L} + \left(4 + \frac{32}{L} \right) \frac{K^{-3/L}}{15}$$

Величины m_L и D_L приведены в [6] и могут быть взяты из следующей таблицы:

L	m_L	D_L
1	0.7326495	0.690
2	1.5374383	1.338
3	2.4016068	1.901
4	3.3112247	2.358
5	4.2354266	2.705
6	5.2177052	2.954
7	6.1962507	3.125
8	7.1836656	3.238

L	m_L	D_L
9	8.1764248	3.311
10	9.1723243	3.356
11	10.170032	3.384
12	11.168765	3.401
13	12.168070	3.410
14	13.167693	3.416
15	14.167488	3.419
16	15.167379	3.421

Для выполнения этого теста требуется длинная последовательность двоичных символов ($n \geq (Q + K)\ L$). Размер блоков L рекомендуется выбирать в пределах $6 < L < 16$.

Размер второго сегмента состоящего из тестируемых блоков, определяется как

$$K = [n/L] - Q \approx 1000 \cdot 2^L.$$

При этом значения величин n, L и Q рекомендуется выбирать следующим образом:

N	L	$Q = 10 \cdot 2^L$
> 387,840	6	640
> 904,960	7	1280
> 2,068,480	8	2560
> 4,654,080	9	5120
> 10,342,400	10	10240
> 22,753,280	11	20480

n	L	$Q = 10 \cdot 2^L$
> 49,643,520	12	40960
> 107,560,960	13	81920
> 231,669,760	14	163840
> 496,435,200	15	327680
> 1,059,061,760	16	655360

Наиболее предпочтительным размером блока следует, очевидно, считать блок с $L = 8$, поскольку расчет остальных вероятностных показателей шифруемых текстов, таких, например, как энтропия H, производится с привязкой к 256-ти символьной кодовой таблице вычислительного устройства.

Примером программной реализации данного теста может быть следующий код.

```
//
=============================================================================
=======
   #include "MourTests.h"
   #include <iostream.h>
   #include <fstream.h>
   #include "MorByt.h"
   #include <string.h>
   #include <stdlib.h>
   #include <string.h>
   #include <stdio.h>
   #include <math.h>
   #include "Fnct.h"
   #include <vcl.h>
   #pragma hdrstop
//
=============================================================================
=======
   #pragma package(smart_init)
   #pragma resource "*.dfm"
   #pragma link "CGAUGES"
// ============================================
         TFrmByt *FrmByt;
// ============================================
    double x, ftu, mss[100], dfo[100];
    double ln2 = 0.6931471805599453094172;
    double sq2 = 1.4142135623730950488016;
    static const double rel_error = 1E-12;
//
=============================================================================
=======
  __fastcall TFrmByt::TFrmByt(TComponent* Owner)
    : TForm(Owner)
  {
  StrC -> Enabled = false;
  Rzlt -> Enabled = false;
  ArgF -> Enabled = false;
  ArgF -> Enabled = false;
  FncE -> Enabled = false;
  }
//
=============================================================================
=======
  void __fastcall TFrmByt::ClszClick(TObject *Sender)
  {
   Close();
  }
//
=============================================================================
=======
  void __fastcall TFrmByt::OpnFClick(TObject *Sender)
  {
      AnsiString MyFName = "";
```

```
    if (OpnD -> Execute())
      {
       MyFName = OpnD -> FileName;
       PthF -> Text = MyFName;
      }
       StrC -> Enabled = true;
  }
//
==========================================================================
=======
  void __fastcall TFrmByt::StrCClick(TObject *Sender)
  {
      char hc;
      unsigned __int8 nm;
      double var = 3.238;
      double mx  = 7.1836656;
      double c, arg, sum, sgm;
      unsigned long int tab[256];
      unsigned int j, ck, dl, md;
      unsigned __int32 k = 3000000;
  // ***************************************************************************
      c = 0.7 - 0.8/(double)8 + (4 + 32/(double)8)*pow(k, -3/(double)8)/15;
      sgm = c * sqrt(var /(double)k);
  // ***************************************************************************
      Gaug -> MaxValue = 100;
      Gaug -> Progress = 0;
      AnsiString whd_name;
      whd_name = PthF -> Text;
      ifstream in  (whd_name.c_str(), ios::in  | ios::binary);
  // ***************************************************************************
    for (ck = 0; ck <= 100; ck ++ )
    {
  // ***************************************************************************
        sum = 0;
    for (j = 0; j <= 255; j ++ )
      {
       tab[j] = 0;
      }
  // ***************************************************************************
    for (j = 1; j <= 3000; j ++ )              // Инициализация массива
      {
       in.get (hc);
       nm = hc;
       tab[nm] = j;
      }
  // ***************************************************************************
    for (j = 3001; j <= 3003000; j ++ )  // Расчет статистических показателей
      {
       in.get (hc);
       nm = hc;
       dl = j - tab[nm];
       sum += (log ((double)dl)) / ln2;
       tab[nm] = j;
      }
  // ***************************************************************************
        ftu = (double)(sum /(double)k);
        arg = fabs((ftu - mx)/(sq2 * sgm));
        Gaug -> Progress = ck;
        mss[ck] = arg;
    }
```

```
// ***************************************************************************
    in.close ();
    StrC -> Enabled = false;
    Rzlt -> Enabled = true;
  }
//
=============================================================================
=======
  void __fastcall TFrmByt::RzltClick(TObject *Sender)
  {
    char hc;                                  // Вывод результатов
    unsigned int i, j;
    double ferfc, ferf;
    unsigned char rzl[100];
    double fnk[100];
  // ***************************************************************************
  //                    Описание таблицы
  // ***************************************************************************
    String str = "   ";
    StrGrd -> FixedRows    =  1; // Определяем количество неподвижных строк
    StrGrd -> ColCount     =  4;        // Определяем количество столбцов
    StrGrd -> RowCount     = 101;              // Задаем количество строк
    StrGrd -> ColWidths[0] =  30;               // Задаем ширину колонки
    StrGrd -> ColWidths[1] = 130;               // Задаем ширину колонки
    StrGrd -> ColWidths[2] = 130;               // Задаем ширину колонки
    StrGrd -> ColWidths[3] =  30;               // Задаем ширину колонки
  // ***************************************************************************
    StrGrd -> Cells [0][0] = " №";             // Задаем название колонки
    StrGrd -> Cells [1][0] = "        X";      // Задаем название колонки
    StrGrd -> Cells [2][0] = "      Erfc";     // Задаем название колонки
    StrGrd -> Cells [3][0] = " R";             // Задаем название колонки
  // ***************************************************************************
        j = 0;
    for ( i = 0; i <= 100; i ++ )
    {
    x = mss[i];
    if (x < 0) ferfc = 2.0 - erfc(-x);
    if (fabs(x) >= 2.2) ferfc = erfc(x);
    if ((x >= 0)&&(fabs(x) < 2.2)) ferfc = 1.0 - erf(x);
    dfo[i] = ferfc;
    if (ferfc <= 0.01)
      {
       rzl[i] = 94; j += 1;
      }
      else
      {
       rzl[i] = 43;
      }
    }
   if (j > 1)
     {
      Lab1 -> Caption = "Тест не выполнен";
     }
   else
     {
      Lab1 -> Caption = "Тест выполнен  Не выполнено тестов " + IntToStr(j);
     }
  // ***************************************************************************
   for ( i = 0; i <= 100; i ++ )
      {
```

```
        hc = (char)rzl[i];
        str[3] = hc;
        StrGrd -> Cells [0][i+1] = IntToStr (i+1);
        StrGrd -> Cells [1][i+1] = " " + FloatToStr (mss[i]);
        StrGrd -> Cells [2][i+1] = " " + FloatToStr (dfo[i]);
        StrGrd -> Cells [3][i+1] = str;
       }
        ArgF -> Enabled = true;
        FncE -> Enabled = true;
        Rzlt -> Enabled = false;
  }
//
=============================================================================
=======
  void __fastcall TFrmByt::ArgFClick(TObject *Sender)
  {
    unsigned int i;
    for ( i = 0; i <= 100; i ++)
       {
        Series1 -> AddXY (i,mss[i],"",clRed);
       }
  }
//
=============================================================================
=======
  void __fastcall TFrmByt::FncEClick(TObject *Sender)
  {
    unsigned int i;
    for ( i = 0; i <= 100; i ++)
       {
        Series2 -> AddXY (i,dfo[i],"",clBlue);
       }
  }
//
=============================================================================
=======
```

Алгоритм тестирования предполагает предварительное создание текстового файла, содержащего сто фрагментов длиной 3003000 байт (3000 – фрагмент инициализации Q и 3000000 – тестируемый фрагмент K). При этом требуется обеспечить независимость каждого из отрезков. Иными словами каждый из них должен быть сформирован с отдельным ключом. Тест считается выполненным, если будет забраковано не более одного из ста фрагментов, поскольку в соответствии с принятой концепцией тестирования [3], показатель P_v не должен быть меньше величины 0.01.

Интерфейс программного продукта разработан таким образом, чтобы можно было ввести путь к файлу, содержащему тестируемую последовательность, и контролировать процесс его обработки. Кроме того обеспечивается отображение числовых и графических результатов тестирования каждого фрагмента. Его внешний вид показан на рис. 2.

Значение функции ошибки $erfc(x)$ производится путем ее вызова из другого программного модуля. Реализация этой процедуры особого значения не имеет. Один из ее вариантов можно найти на сайте NIST по адресу http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

В ***заключение*** следует отметить, что испытания различных шифров, проведенные с использованием этого теста, показывают его крайнюю чувствительность к малейшей неравномерности распределения вероятностей символов. Даже незначительные отклонения тестирующей функции f_n от ее теоретически ожидаемого выборочное среднее значения m_L,

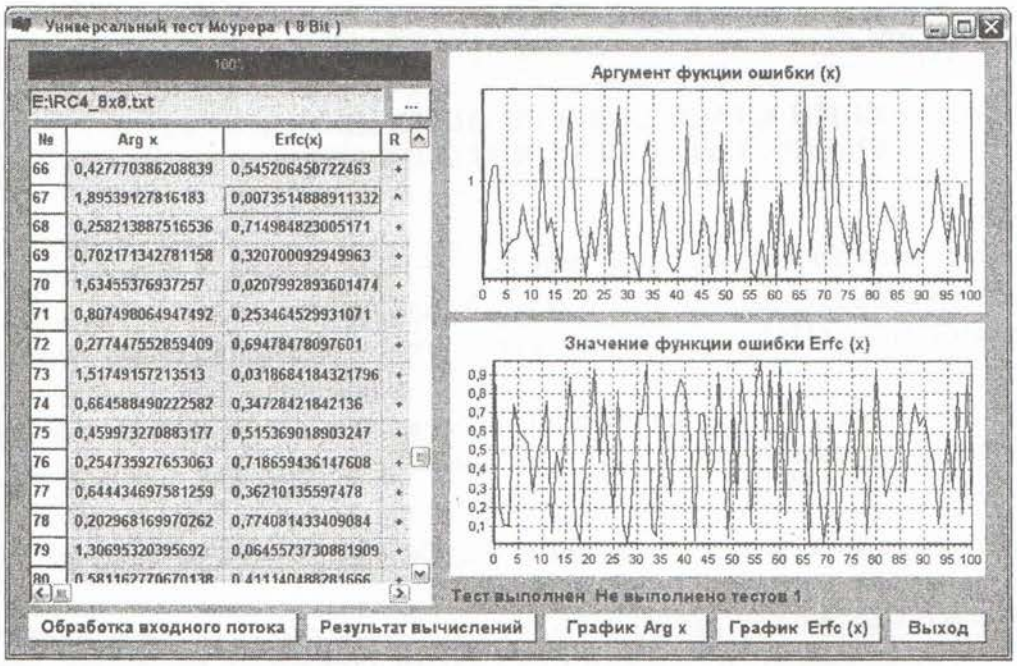

Рис.2. Внешний вид интерфейса

ведет к непризнанию тестируемого отрезка последовательности «случайной». С учетом этого можно сделать вывод об эффективности универсального теста.Его выполнение может служить основанием для роста уверенности в качестве разрабатываемого генератора ПСП.

Список литературы

1. eSTREAM, the ECRYPT Stream Cipher Project. http://www.ecrypt.eu.org/ stream/index.html

2. Клод Шеннон. Теория связи в секретних системах «Работы по теории информации и кибернетике», М., ИЛ, 1963, с. 333-369

3. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. NIST Special Publication 800-22. May 15, 2001.

4. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness//http://www.stat.fsu.edu/pub/diehard/

5. Statistical test suite Crypt-X //http://www.isi.qut.edu.au/resources/cryptx/

6. A Universal Statistical Test for Random Bit Generators. Ueli M. Vaurer. Appeared in Jornal of Cryptology, vol. 5, no. 2, 1992, pp. 89-105.

Предложена программная реализация универсального статистического теста Маурера для псевдослучайных последовательностей, применяемых в криптографии и дано его теоретическое обоснование.
Ключевые слова: ПСП, криптографическая система, тестирование

Запропонована програмна реалізація універсального статистичного тесту Маурера для псевдовипадкових послідовностей, які використовуються в криптографії, та дано його теоретичне обґрунтування.
Ключові слова: ПСП, криптографічна система, тестування

Programmatic realization of universal statistical test of Maurer is offered for pseudocasual sequences, applied in cryptography and his theoretical ground is given.
Keywords: pseudocasual sequences, cryptographic system, testing

Рецензент: Скрипник Л.В.
Надійшла 27.08.2010