УДК 004.056.55

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ЯЗЫКОВ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИ МОДЕЛИРОВАНИИ МЕТОДОВ КРИПТОГРАФИЧЕСКИХ ПРЕОБРАЗОВАНИЙ

Е.Г. КАЧКО, Д.К. ТЕЛЕВНЫЙ

Статья посвящена проблеме возможности моделирования методов криптографических преобразований в контексте функционального программирования. Проанализированы основные особенности чисто функциональных языков, а также математический базис симметричных шифров. Выявлены и обоснованы преимущества и недостатки функционального программирования при реализации существующих алгоритмов криптографических преобразований, моделировании и верификации создаваемых. На основе предлагаемого исследования авторами приведены доказательства о целесообразности использования такого подхода как инструмента при моделировании и верификации преобразований.

Ключевые слова: функциональный язык, Haskell, C++, симметричные шифры, AES.

ВВЕДЕНИЕ

Для современных криптографических алгоритмов к условиям хорошего алгоритма относят не только открытость, криптостойкость, но и возможность легкой реализации на различных программных и аппаратных архитектурах. Так одним из способов повышения быстродействия является распараллеливание вычислений на мульти- и многопроцессорной конфигурации. [1]

Отличным выбором при моделировании распараллеливаний новых методов является использование функциональных языков для более точного описания возможности мультипоточного выполнения при имплементации алгоритма на разных языках. В работе исследуется возможность использования функциональных языков на примере криптографического алгоритма AES.

1. ОСНОВНАЯ ИДЕЯ ИССЛЕДОВАНИЯ

Парадигма функционального программирования трактует процесс вычисления функций в математическом понимании последних, в отличии от понимания функции как подпрограммы в процедурном программировании.

Некоторые подходы программирования специфичны только функциональной парадигме и чужды другим подходам (процедурному и особенно ООП). Но учитывая то, что большинство современных языков реализуют гибридные парадигмы, возможно использование особенностей ФП.

К особенностям ФП можно отнести:

– Представление функции в ЯП – в виде математических функций высших порядков, т.е. основная идея состоит в том, что функция не отличается от других обьектов данных. Это значит что ФВП могут принимать функции в качестве аргументов и возращать функцию в качестве результата. Подходу возможен блягодаря использованию каринга (каррирования) предложенного Хаскелом Керри.

– Использование чистых функций как таких, что не имеют побочных эффектов на состояние процесса выполнения. Такие функции легко могут быть мемоизированы (результаты вычислений заносятся в таблицу и при повторном вызове подставляются в виде константы). Именно эти функции позволяют ценой небольшого увеличения расхода памяти оптимизировать процесс выполнения.

– Рекурсия является одним из возможных средств организации цикла в ФЯП (в идее ФП отсутствует понятие цикла), может потребоваться увеличение стека вызовом, что можно обойти используя хвостовую рекурсию.

Как говорилось раннее pure функции, являясь независимыми от состояния программы, позволяют оптимизировать код в различных потоках. Любая такая функция, не имея побочных эффектов, может быть представлена в виде абстрактного листа дерева выполнения, где количество листьев в узле – потоки, которые могут выполнятся независимо. Кроме этого аргументом функции может быть ссылка на другую функцю. [2]

Далее приведен пример того, как правильное применение идеи функционального программирования может помочь при моделирование новых криптографических преобразований.

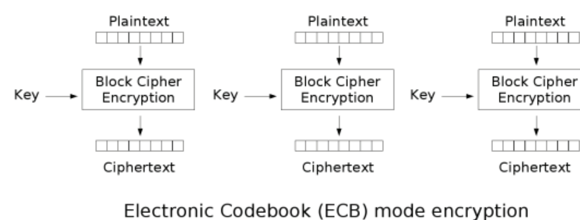

Рис. 1. Схема режима электронной книги

Начать следует с того, что любая криптографическая функция (допустим симметричный шифр) является математической функцией.

$$C = E(P, K)\,; \quad (1.1)$$

$$P = D(C, K)\,. \quad (1.2)$$

Формулы (1.1) и (1.2) представляют собой функции из двух параметров с возвращаемым значением. С точки зрения ФП, E и D – функции высшего порядка, где P – блок открытого текста (длина зависит от шифра), C – блок шифротекста, K – криптографический ключ. Функции (1.1) и (1.2) являются также чистыми (англ. Pure), что значит они не обладают состоянием и контекстно-безопасны.

Рассмотрим это на примере режимов шифрования. Каждый режим представляет собой функцию над массивом блоков (блочных шифров) данных (допустим выравнивание и дописывание до кратности уже сделано) и ключа. Также встречается применение вектора инициализации в сцепленных режимах.

Режим ECB (режим электронной книги) не предполагает сцепления блоков. На рис.1.1 изображена схема режима электронной книги. [3]

Так, преобразование в этом режиме (используя симметричный шифр в качестве параметра) можно изобразить в виде функции высшего порядка.

$$C = ECB(E, P, K)\,; \quad (1.3)$$

$$P = ECB(D, P, K)\,. \quad (1.4)$$

где E, D – функции (1.1) и (1.2), P – открытый текст, C — шифротекст, K – симметричный ключ. Стоит учитывать, что расписание ключей для этого режима одинаково для всех данных и в императивных языках его стоит вычислять перед началом шифрования для более быстрого доступа. С точки зрения ФП (1.3) – pure, поэтому K может быть не только массивом из слов или байт, но и функцией.

Давайте рассмотрим реализацию этого режима на языке F#. Далее приведен код объявления этой функции и ее тела. Пусть уже задан симметричный шифр в виде функции заглушки, принимающий два аргумента и возвращающий значение зашифрованного блока.

```
let aesEncipher (cipherKey: int array) (plaintext: int array) =
    //cryptotransformation over one block
```

Аргумент cipherKey может быть смело заменен функцией, возвращающей ключ для каждого раунда. Не стоит забивать, что функции ФЯ могут быть каррированы и декапрированы.

Так общая сигнатура функции генерации расписания ключа выглядит как:

```
let rec scheduleKey (key: int array) (numberOfRound: int) =
    // return new key for specific round
```

Рассмотрим реализацию режима ECB. В приведенном коде в качестве первого параметра представлено объявление отображения функции. Таким образом в качестве этого параметра может быть передана как заранее определенная функция, так и лямбда-функция.

```
let ecbMode (f: int array → int array → int array) (plaintext: int array array) (key: int array) =
    let curriedKeyFunc = scheduleKey key
    let curriedCipher = f curriedKeyFunc
   Parallel.map curriedCipher plaintext;
```

Функция curriedCipher будет применена ко всем элементам plaintext. Чтобы распараллелить обработку массива, нужно место array.map вызвать Parallel.map.

2. МОДЕЛЛИРОВАНИЕ ШИФРА RIJNDAEL В ФУНКЦИОНАЛЬНОМ КОНТЕКСТЕ

Представим реализацию основного раунда AES-128 в Haskell и реализация на C.

В Haskell обозначим представление, объявленное в следующем виде и представленное, как список списков 8-битных слов (байт). Благодаря декларативным возможностям функциональных языков, функция aesMainRound, как показано ниже, реализована композицией составных функций преобразований.

```
type State = [[Word8]]
aesMainRound :: State -> State -> State
aesMainRound rk = addRoundKey rk .
                  MixColumns. shiftRows .
                  subBytes
```

В языке C данный полный алгоритм шифрования блока выглядит следующим образом

```
AddRoundKey(0, state);
for (unsigned round = 1; round < Nr; round++){
   SubBytes(state);
   ShiftRows(state);
   MixColumns(state);
   AddRoundKey(round, state);
}
SubBytes(state);
ShiftRows(state);
AddRoundKey(Nr, state);
```

Рассмотрим функцию реализацию subBytes в этих языках. Так, в Haskell данная функция, определена таким образом.

```
subBytes :: State -> State
subBytes = map row
           where row = map subByte

subByte :: Word8 -> Word8
subByte x = sbox ! x

sbox :: Array Word8 Word8
sbox = array (0, 255) $ zip [0..] vals
     where vals = [ 0x63, 0x7c, ...,
0x16 ]
```

Функция subBytes использует один из подходов в функциональных языках, известный как маппинг. Так, map для всех элементов списка State вызывает функцию row, которая объявлена внутри subBytes, where – ключевое слово для определения локальных функций. Таким образом из кода видно, что функции находятся в суперпозиции, обьявленной декларативно. Функция zip – объявлена в библиотеке. Результатом этой функции является список пар элементов двух входящих списков. $ – правоассоциативный оператор приложения, в данном контексте подставляет значение возвращаемое, zip в array. ! - оператор индексирования.

В языке С данная реализация выглядит следующим образом

```
// case for matrix of bytes
unsigned char* t =(unsigned
char*)state;
for (int i = 0; i < 16; i++){
   t[i] = sbox[t[i]];
}

// case for 32word array
unsigned value, result, temp;
for (auto i = 0; i < Nb; i++){
    temp = state[i];
   value = 0xff & temp;
   result = sbox[value];
   value = 0xff & (temp » 8);
   result |= (unsigned)sbox[value] « 8;
   value = 0xff & (temp » 16);
   result |= (unsigned)sbox[value] «16;
   value = 0xff & (temp » 24);
   result |= (unsigned)sbox[value] «24;
   state[i] = result;
}
```

В обеих вариантах массив подстановок уже определен заранее для оптимизации. В случае с AES, где эти значения предопределены, это оправдано. Однако для других алгоритмов, где используется другая схема генерации блока постановок, необходимо использовать функцию генерации. В Haskell это делается заменой параметра в функции zip. [4]

Другим важным этапом в SP – модели является линейное преобразование перестановкой. Можно привести две реализации заданного этапа.

```
shiftRows :: State -> State
shiftRows xss = [ shift n xs | (n,
xs)<-
          zip [0..] xss ]
          where shift n xs =
                drop n xs ++ take n xs

shiftRows :: State -> State
shiftRows  [[a1, a2, a3, a4],
             [b1, b2, b3, b4],
             [c1, c2, c3, c4],
             [d1, d2, d3, d4]] =
             [[a1, a2, a3, a4],
             [b2, b3, b4, b1],
             [c3, c4, c1, c2],
             [d4, d1, d2, d3]]
```

Первая функция достает каждую строку из xss, выполняем zip с функцией shift для промежутка значений от 0 до 3. Shift проводит циклический сдвиг.

Второй вариант выполнен в декларативном стиле и использует подход, известный как pattern matching. Такой подход потребляет больше памяти, однако его можно использовать, как наглядный пример в модульном тестировании при верификации частей алгоритма.

В языке С данная функция выглядит следующим образом.

```
 unsigned temp;
//shifting 2st row
 temp = state[1];
 state[1] =(temp » 8) | (state[1] «
24);
//shifting 3nd row This can be modified
by
//adding XOr oper to 2 opernads;
 temp = state[2];
 state[2] = (temp « 16)|(state[2] »
16);
//shifting 4th row;
 temp = state[3];
 state[3] = (temp » 24)|(state[3] « 8);
```

В приведенном выше коде показано, что для циклического сдвига используются побитовые операции.

Функция mixColumns является камнем преткновения в данном шифре. Ее трудно реализовывать в обеих языках, однако Haskell позволяет упростить процесс за счет декларативных возможностей. Пример функции приведен ниже.

```
mixColumns :: State -> State
mixColumns = transpose . map mixColumn
. Transpose

mixColumn :: [Word8] -> [Word8]
mixColumn [a0, a1, a2, a3] =
    [b0, b1, b2, b3]
    where b0 = mult2 ! a0 'xor' mult3
A1
```

```
                  `xor` a2 `xor` a3

              b1  =  a0  `xor`  mult2  !  a1
`xor`
                  mult3 ! a2 `xor` a3

              b2 = a0 `xor` a1 `xor` mult2
!
                  a2 `xor` mult3 ! a3

              b3  =  mult3  !  a0  `xor`  a1
`xor`
                  a2 `xor` mult2 ! a3

   mult2, mult3 :: Array Word8 Word8
   mult2 = array (0, 255) $ zip [0..] vals
         where vals =[0x00,0x02 ..., 0xe5
]

   mult3 = array (0, 255) $ zip [0..] vals
          where  vals  =  [0x00,  0x03  ..
0x1a]
```

Определение mixColumns демонстрирует преимущества функционального похода.

В языке С данное преобразование имеет следующую реализацию:

```
   #define xtime(x)((x«1) ^ (((x»7) & 1) *
0x1b))
   #define Multiply(x,y) (((y & 1) * x) ^
((y » 1 & 1) * xtime(x)) ^ ((y » 2 & 1) *
xtime(xtime(x)))  ^  ((y  »  3  &  1)  *
xtime(xtime(xtime(x)))) ^ ((y » 4 & 1) *
xtime(xtime(xtime(xtime(x))))))

   void MixColumns(unsigned* state)
   {
      int i;
      unsigned char* st;
      unsigned char Tmp, Tm, t;
      for (i = 0; i < 4; i++)
      {
         st = (unsigned char*)state;
          t = st[i];
         Tmp = st[i] ^ st[4 + i] ^
               st[8 + i]  ^ st[12 + i];
          Tm = st[i] ^ st[4 + i];
          Tm =  xtime(Tm);
          st[i] ^= Tm ^ Tmp;
          Tm = st[4 + i] ^ st[8 + i];
          Tm = xtime(Tm);
          st[4 + i] ^= Tm ^ Tmp;
          Tm = st[8 + i] ^ st[12 + i];
          Tm = xtime(Tm);
          st[8 + i] ^= Tm ^ Tmp;
          Tm = st[12 + i] ^ t;
          Tm = xtime(Tm);
          st[12 + i] ^= Tm ^ Tmp;
       }
    }
```

3. ПРЕИМУЩЕСТВА ФУНКЦИОНАЛЬНОГО ПОДХОДА С ТОЧКИ ЗРЕНИЯ БЕЗОПАСНОСТИ

Доказано, что, и функциональные языки, и императивные являются Тьюринг-полными, т.е. множество вычислений включает множество, которое использует Тьюринг-машина. Тем не менее это свойство не значит, что программа безопасна. Пять принципов функциональных языков предоставляют функции, которые заставлют или улучшают безопасность ПО, построенного на этих языках.

3.1 Функции с параметрами и результатами

Все процедуры в функциональных языках представляют из себя функции и четко выделяют входящие значения от возращаемых. Функции следуют моделям функций в математической теории: параметры пренадлежат определенному домену значений, а результаты находятся в определенном множестве. Следовательно – функция это математическое отображение параметров на значение. Это заставляет функциональное программирование реализовывать вычисления таким образом, что вне зависимости от порядка выполнения выражений программа возращает один и тот же результат для разных параметров. Это обеспечивает безопасное выполнение с детерминированым поведением, обеспечиваемым строго алгоритмом, описаным программно.

3.2 Привязка параметров

Функциональная парадигма не имеет понятия присваивания значений переменной. Эта операция имеет соответствующую команду загрузки в ассемблерных языках. Такие действия тесно связывают программу и аппаратное обеспечение. Хотя некоторые языки 3-го поколения имеют ограничения на присваивание, но оно имеет значение лишь для компилятора, но не для ОС. Следовательно языки содержащие такие конструкции, не могут полностью контролировать значение, т.е. содержимое памяти или отслеживать изменения значения другим источником.

Чисто функциональные языки напротив не имеют синтаксических и семантических средств для присваивания. Параметры, видимые в теле функции – всего лишь аргументы и любые другие локальные параметры, созданные через «where» or «let» clause. В сущности параметры содержат константы, которые только применяются к выражению и не изменяются, пока функция не завершается. Как только параметр в теле функции привязан к значению — оно сохраняется до конца жизни вызова функции. Таким образом ведут себя и параметры функции. При следующем вызове функции параметры не содержат информации о ранних вызовах и не могут менятся пока не закончится выполнение. Это обеспечивает безопасное программирование с гарантией, что ни какие изменения в переменных окружения, не

связанных с параметрами, не могут повлиять на результат.

3.3 Рекурсия

Хвостовая рекурсия позволяет алгоритму быть оптимизированным компилятором. При применении рекурсивных вызовов безопасность обеспечивается на основе доказательной техники, основанной на математической индукции.

3.4 Ссылочная прозрачность

Все функции являются ссылочно-прозрачными. Это значит, что любая функция возвращает один и тот же результат для одинакового набора параметров независимо от контекста вызовов. Это также значит что порядок вычисления аргуметов не влияет на результат. Это позволяет инкапсулировать целый алгоритм в функцию, зная что никакое внешнее внешенее влияние прямо или косвенно затронет результат.

3.5 Первичные функции

При таком подходе функции могут быть не только переданы в качестве параметра, но и возвращаемы как результат. Это позволяет безопасному программированию создавать программы, которые моделируют математическую композицию. Программа в этом случае становится большим математическим вычислением.

Таким образом при правильном применении функциональной парадигмы основные криптографические аттаки могут быть выполнены только используя уязвимость ресурсов, на которых выполняется программа (уязвимости ОС, уязвимости в аппаратном обеспечении). [5]

ЗАКЛЮЧЕНИЕ

В работе была изучена возможность использования функционального похода для разработки и моделирования блочных шифров. Имплементация алгоритма на Haskell позволяет упростить процесс, сделав разработку более наглядной и соответствующей математической базе данного алгоритма. Разработка приложения в функциональной парадигме позволяет упростить процесс нахождения возможностей оптимизации программного кода. Кроме этого такой подход делает разработку более гибкой – конструирование новых функций на базе уже существующих.

В дальнейшем планируется проводить разработку на более низком уровне математической модели. Такой подход позволит произвести оптимизацию выполнения путем объединения составных частей преобразования.

Литература

[1] *Рябко Б. Я., Фионов А. Н.* Криптографические методы защиты информации [Текст].— М.;— Изд-во Горяч.Линия-Телеком, 2005.

[2] *А. Филд, П. Харрисон* Функциональное программирование: Пер. с англ. — М.: Мир, 1993. — 637 с, ил. ISBN 5-03-001870-0. Стр. 120 [Глава 6: Математические основы: λ-исчисление].

[3] NIST Recommendation for Block Cipher Modes [Электронный ресурс]. – Режим доступа: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf – 16.11.2001 г. – Загл. с экрана.

[4] Cryptographic Block Ciphers in Functional Programming: A Case Study on Feldspar and AES [Электронный ресурс].– Режим доступа: https://gitlab.com/gregor_ulm/dat085_feldspar/blobmaster/GregorUlm.FinalReport.May31.pdf – 26.06.2010 г. – Загл. с экрана.

[5] *Арто Мустайоки*. Теория функционального синтаксиса. От семантических структур к языковым средствам [Текст]. – Litres, 2014.

Качко Елена Григорьевна, кандидат технических наук, профессор кафедры ПИ ХНУРЭ. Научные интересы: криптография, криптоанализ, параллельные вычисления.

Телевный Дмитрий Константинович, студент группы ИПЗм-15-1 факультета КН ХНУРЭ. Научные интересы: применение функционального подхода в разработке систем, методы симметричного криптопреобразования.

УДК 004.056.55

Дослідження можливості використання мов функціонального програмування для моделювання методів криптографічних перетворень / О.Г. Качко, Д.К. Телевний. // Прикладна радіоелектроніка: наук.-техн. журнал. – 2016. – Том 15, № 3. – С 162 – 166.

Стаття присвячена дослідженню можливості моделювання методів криптографічний перетворень в контексті функціонального програмування. На основі проведеного дослідження автори показали доцільність використання такого підходу, як інструменту під час моделювання та верифікації перетворень.

Ключові слова: функціональна мова, Haskell, C++, симетричні шифри, AES

Іл.: 01. Бібліогр.: 05 найм.

UDC 004.056.55

Studying the possibility of using functional programming languages in modelling methods of cryptographic transformations / O.G. Kachko, D.K. Televnyi // Applied Radio Electronics: Sci. Journ. – 2016. – Vol. 15, № 3. – P. 162 – 166.

The paper is devoted to the possibility of modelling methods of cryptographic transformations in functional programming context. The main peculiarities of purely functional languages as well as the mathematical basis of symmetric ciphers are analysed. Advantages and disadvantages of functional programming at realizing the existing algorithms of cryptographic transformations, modeling and verifying the ones being created have been revealed and substantiated. On the basis of the proposed study the author have presented evidence of the feasibility of such an approach as a tool for modelling and verifying transformations.

Keywords: functional language, Haskell, C++, symmetric ciphers, AES.

Fig.: 01. Ref.: 05 items.