**Лекция 13. Методы создания алгоритмически безопасных процедур. Методы создания самотестирующихся и самокорректирующихся программ для решения вычислительных задач**

**Методы создания алгоритмически безопасных процедур.**

Основной отличительной особенностью подхода, связанного с созданием алгоритмически безопасного ПО является то, что начало процесса обеспечения безопасности программ при их разработке можно перенести на более ранние этапы жизненного цикла программного обеспечения, например, на этапы, предшествующие этапу испытания программ, тем самым увеличив общее время на внесение в программы защитных функций.

Одно из основных достоинств методов создания алгоритмически безопасного программного обеспечения заключается в том, что данные методы позволяют защищать программное обеспечение, как на этапе разработки, так и на этапе его эксплуатации.

Основная сложность в разработке безопасных программ указанного типа заключается в трудности нахождения эффективных алгоритмов их функционирования, которые являлись к тому же доказуемо корректными. И, предположительно, проблема неразрешимости доказательства безопасности для сложных программ при использовании методов их тестирования на этапе испытаний остается, в случае применения методов создания алгоритмически безопасного программного обеспечения, по-прежнему неразрешимой, что будет видно из дальнейших рассуждений.

Одним из главных методических вопросов создания безопасного программного обеспечения является постановка задачи (формулировка проблемы) ее разработки. Предположим, некоторому разработчику (коллективу разработчиков) предписывается разработать программу P для некоторого объекта автоматизации. При этом последствия некорректного функционирования программы таковы, что могут привести к неким "нежелательным" или даже катастрофическим последствиям для объекта автоматизации. Исходя из гипотезы, что в общем случае проблема доказательства безопасности для сложных комплексов программ является неразрешимой, неформальная качественная постановка задачи разработки указанных программ может быть тогда сформулирована следующим образом.

*Постановка 1.* "При некоторых условиях и ограничениях необходимо разработать программу *P*, которая корректно вычисляет результат *почти* для всех своих входных значений".

Таким образом, разработчик констатирует факт, что лишь для незначительной (возможно достаточно малой) доли входных значений программа может выдать некорректное выходное значение в результате необнаруженного программного дефекта или активизации необнаруженной программной закладки.

При использовании контрольно-испытательных методов анализа безопасности программ, когда анализу подвергается только исполняемый код программы можно получить (как правило, экспертным путем) либо приближенные количественные характеристики обнаружения дефектов в контролируемых программах, либо стратификационные характеристики ненарушения программой некоторых условий безопасности. В этом случае постановка 1 задачи разработки программы *P* принципиально не меняется. (Естественно не рассматриваются простейшие случаи, когда при тестировании возможен контроль результата программы при полном переборе всех входных значений, при всех ограничениях и допущениях и т.п.).

Постановка же задачи разработки безопасного программного обеспечения за счет алгоритмически безопасных процедур меняется с точки зрения получения точных количественных (в данном случае вероятностных) характеристик кардинальным образом.

*Постановка 2.* "При некоторых условиях и ограничениях необходимо разработать программу *P\**, которая корректно вычисляет результат для всех своих входных значений с *пренебрежимо малой* вероятностью ошибки".

На примере постановок 3, 4 и 5, 6 можно также проследить существенную разницу в парадигме разработки безопасных программ. Постановки 3 и 4 в отличие от постановок 1 и 2 учитывают не "содержание" наборов входных значений программы P, а их распределения вероятностей, а постановки 5 и 6 являются в некотором смысле обобщенными и учитывают заданный структурный критерий тестирования.

*Постановка 3.* "При некоторых условиях и ограничениях необходимо разработать программу *P*, которая некорректно вычисляет результат *лишь для некоторого* частного распределения вероятностей своих входных величин".

*Постановка 4.* "При некоторых условиях и ограничениях необходимо разработать программу *P\**, которая корректно вычисляет результат для любого распределения вероятностей своих входных величин с пренебрежимо малой вероятностью ошибки".

*Постановка 5.*"При некоторых условиях и ограничениях необходимо разработать программу *P*, которая корректно вычисляет результат почти на всех тестах полной системы тестов программы относительно заданного структурного критерия тестирования".

*Постановка 6.* "При некоторых условиях и ограничениях необходимо разработать программу *P\**, которая корректно с пренебрежимо малой вероятностью ошибки вычисляет результат на всех тестах полной системы тестов относительно заданного структурного критерия тестирования".

Одним из принципиальных условий решения задачи в постановке 2, 4 и 6 является наличие некоего свойства алгоритмической трансформации, позволяющего переходить от традиционного пути создания программ, которые будут затем проверяться на наличие дефектов, к априорно безопасным программам. К числу таких примеров можно отнести самокорректирующиеся и самотестирующиеся программы, обладающие свойством случайной самосводимости и программы, созданные на базе методов конфиденциальных вычислений. Рассмотренные до этого методы анализа безопасности ПО связаны с попытками обезопасить фактически уже разработанное программное обеспечение от действий злоумышленника. Это означает, что разработка безопасного ПО возможна за счет создания средств противодействия программным закладкам для продуктов, созданных на основе существующих информационных технологий создания программного обеспечения. То есть, только после факта разработки программ начинается верификационный анализ, тестирование или контроль их на технологическую безопасность. В этом смысле проблема обеспечения технологической безопасности программного обеспечения более близка к фундаментальной проблеме его надежности.

В то же время, данные методы создания безопасного ПО характеризуются рядом существенных недостатков, к числу которых можно отнести: невозможность перекрытия для большинства программных комплексов всего спектра тестовых наборов исходных данных; необходимость создания высоконадежных механизмов тестирования программ, например, механизмов экстраполяции результатов испытаний; существенные ресурсозатраты на проведение испытаний и т.п.

Поэтому в последнее время появилась насущная необходимость в создании новых информационных технологий разработки ПО, исходно ориентированных на создание безопасных программных продуктов относительного заданного класса. В этом случае проблема исследований сводится к разработке таких математических моделей, которые представляются адекватной формальной основой для создания методов защиты программного обеспечения на этапе его проектирования и разработки. При этом изначально предполагается, что:

* один или несколько участников проекта являются (или, по крайней мере, могут быть) злоумышленниками;
* в процессе эксплуатации злоумышленник может вносить в программы изменения (необязательно связанные с внедрением апостериорных программных закладок);
* средства вычислительной техники, на которых выполняются программы, не свободны от аппаратных закладок.

Тогда, исходя из этих допущений, формулируется следующая неформальная постановка задачи: "Требуется разработать программное обеспечение таким образом, что несмотря на указанные выше "помехи" оно функционировало бы правильно". Одно из основных достоинств здесь состоит в том, что одни и те же методы позволяют защищаться от злоумышленника, действующего как на этапе разработки, так и на этапе эксплуатации программного обеспечения. Однако, это достигается за счет некоторого замедления вычислений, а также повышения затрат на разработку программного обеспечения.

В рамках указанного выше подхода на данный момент известны два направления:

#### Методы создания самотестирующихся и самокорректирующихся программ для решения вычислительных задач

#### Методы теории конфиденциальных вычислений

**Общие принципы создания двухмодульных вычислительных процедур и методология самотестирования**

Пусть необходимо написать программу *P* для вычисления функции *f* так, чтобы *P*(*x*)=*f*(*x*) для всех значений *x*. Традиционные методы верификационного анализа и тестирования программ не позволяют убедиться с вероятностью близкой к единице в корректности результата выполнения программы, хотя бы потому, что тестовый набор входных данных, как правило, не перекрывают весь их возможный спектр. Один из методов решения данной проблемы заключается в создании так называемых самокорректирующихся и самотестирующихся программ, которые позволяют оценить вероятность некорректности результата выполнения программы, то есть, что *P*(*x*)*f*(*x*) и корректно вычислить *f*(*x*) для любых *x*, в том случае, если сама программа *P* на большинстве наборах своих входных данных (но не всех) работает корректно.

Чтобы добиться корректного результата выполнения программы *P*, вычисляющей функцию *f*, нам необходимо написать такую программу *Tf*, которая позволяла бы оценить вероятность того, что *P*(*x*)*f*(*x*) для любых *x*. Такая вероятность будет называться *вероятностью ошибки* выполнения программы *P*. При этом *Tf* может обращаться к *P* как к своей подпрограмме.

Обязательным условием для программы *Tf* является ее принципиальное *временное отличие* от любой корректной программы вычисления функции *f*, в том смысле, что время выполнения программы *Tf*, не учитывающее время вызовов программы *P*, должно быть значительно меньше, чем время выполнения любой корректной программы для вычисления *f*. В этом случае, вычисления согласно *Tf* некоторым количественным образом должны отличаться от вычислений функции *f* и *самотестирующаяся программа* может рассматриваться как независимый шаг при верификации программы *P*, которая предположительно вычисляет функцию *f*. Кроме того, желательное свойство для *Tf* должно заключаться в том, чтобы ее код был насколько это возможно более простым, то есть *Tf* должна быть *эффективной* в том смысле, что время выполнения *Tf* даже с учетом времени, затраченного на вызовы *P* должно составлять константный мультипликативный фактор от времени выполнения *P*. Таким образом, самотестирование должно лишь незначительно замедлять время выполнения программы *P*.

Пусть π - означает некоторую вычислительную задачу и/или некоторую задачу поиска решения. Для *x*, рассматриваемого в качестве входа задачи, пусть π(*x*) обозначает результат решения задачи π. Пусть *Р* – программа (предположительно предназначенная) для решения задачи π, которая останавливается (например, не имеет зацикливаний) на всех входах задачи π. Будем говорить, что *Р имеет дефект*, если для некоторого входа *x* задачи π имеет место *P*(*x*)π(*x*).

Определим (*эффективный*) *программный чекер C*π для задачи π следующим образом. Чекер *C*π*P*(*I*,*k*) – является произвольной вероятностной машиной Тьюринга, удовлетворяющей следующим условиям. Для любой программы *P* (предположительно решающей задачу π), выполняемой на всех входах задачи π, для любого элемента *I* задачи π и для любого положительного *k* (параметра безопасности) имеет место:

• если программа *P* не имеет дефектов, т.е. *P*(*x*)=π(*x*) для всех входов *x* задачи π, тогда *C*π*P*(*I*,*k*) выдаст на выходе ответ «Норма» с вероятностью не менее 1-1/2*k*;

• если программа *P* имеет дефекты, т.е. *P*(*x*)π(*x*) для всех входов *x* задачи π, тогда *C*π*P*(*I*,*k*) выдаст на выходе ответ «Сбой» с вероятностью не менее 1-1/2*k*.

*Самокорректирующаяся программа* – это вероятностная программа *Cf*, которая помогает программе *P* скорректировать саму себя, если только *P* выдает корректный результат с низкой вероятностью ошибки, то есть для любого *x*, *Cf* вызывает программу *P* для корректного вычисления *f*(*x*), в то время как собственно сама *P* обладает низкой вероятностью ошибки.

*Самотестирующейся/самокорректирующейся программной парой* называется пара программ вида (*Tf*,*Cf*). Предположим, что пользователь может взять любую программу *P*, которая целенаправленно вычисляет *f* и тестирует саму себя при помощи программы *Tf*. Если *P* проходит такие тесты, тогда по любому *x*, пользователь может вызвать программу *Cf*, которая, в свою очередь, вызывает *P* для корректного вычисления *f*(*x*). Даже если программа *P*, которая вычисляет значение функции *f* некорректно для некоторой небольшой доли входных значений, ее в данном случае все равно можно уверенно использовать для корректного вычисления *f*(*x*) для любого *x*. Кроме того, если удастся в будущем написать программу *P*′ для вычисления *f*, тогда некоторая пара (*Tf*,*Cf*) может использоваться для самотестирования и самокоррекции *P*′ без какой-либо ее модификации.

Вероятностная программа *M* является *вероятностной оракульной программой*, если она может вызывать другую программу, которая является исполнимой во время выполнения *M*. Обозначение MAозначает, что *M* может делать вызовы программы *A*.

Пусть *P* - программа, которая предположительно вычисляет функцию *f*. Пусть *I* является объединением подмножеств *In*, где *n* принадлежит множеству натуральных чисел *N* и пусть *Dp*={*Dn*|*n*∈*N*} есть множество распределений вероятностей *Dn* над *In*. Далее, пусть *err*(*P*,*f*,*Dn*) – это вероятность того, что *P*(*x*)*f*(*x*), где *x* выбрано случайно в соответствии с распределением *Dn* из подмножества *In*. Пусть β - есть некоторый параметр безопасности. Тогда (ε1,ε2)-*самотестирующейся программой* для функции *f* в отношении *Dp* с параметрами 0≤ε1<ε2 ≤1 - называется вероятностная оракульная программа *Tf*, которая для параметра безопасности β и любой программы *P* на входе *n* имеет следующие свойства:

* если *err*(*P*,*f*,*Dn*)≤ε1, тогда программа *TfP* выдаст на выходе ответ «норма» с вероятностью не менее 1-β.
* если *err*(*P*,*f*,*Dn*)≥ε2, тогда программа *TfP* выдаст на выходе «сбой» с вероятностью не менее 1-β.

Оракульная программа *Cf* с параметром 0≤ε<1 называется ε*-самокорректирующейся программой* для функции *f* в отношении множества распределений *Dp*, которая имеет следующее свойство по входу *n*, *x*∈*In* и β. Если *err*(*P*,*f*,*Dn*)≤ε, тогда *CfP*=*f*(*x*) с вероятностью не менее 1-β.

(ε1,ε2,ε) - *самотестирующейся/ самокорректирующейся программной парой* для функции *f* называется пара вероятностных программ (*Tf*,*Cf*) такая, что существуют константы 0≤ε1<ε2≤ε<1 и множество распределений *Dp* при которых *Tf* -есть (ε1,ε2)-самотестирующаяся программа для функции *f* в отношении *Dp* и *Cf* - есть ε*-*самокорректирующаяся программа для функции *f* в отношении распределения *Dp*.

*Свойство случайной самосводимости.* Пусть *x*∈*In* и пусть *c*>1 - есть целое число. Свойство случайной самосводимости заключается в том, что существует алгоритм *A*1, работающий за время пропорциональное *nO(1)*, посредством которого функция *f*(*x*) может быть выражена через вычислимую функцию *F* от *x*, *a*1,...,*ac* и *f*(*a*1),...,*f*(*ac*) и алгоритм *A*2, работающий за время пропорциональное *nO(1)*, посредством которого по данному *x* можно вычислить *a*1,...,*ac*, где каждое *ai* является случайно распределенным над *In* в соответствии с *Dp*.

**Устойчивость, линейная и единичная состоятельность**

Пусть свойство*I* выражается уравнением *I*(*x*1,...,*xk*)=0, где кортеж (〈*x*1,...,*xk*) выбирается с распределением *E* из пространства *Dk*. Пара (*I*,*E*) характеризует семейство функций *F*, где *f*∈*F* тогда и только тогда, когда для всех (*x*1,...,*xk*) с ненулевой выборкой элементов кортежа из*E*, *I f*(*x*1,...,*xk*)=0. Базовой техникой самотестирования является идентификация свойства *устойчивости* для семейства функций *F*. Неформально (*D*,*D*′)- *устойчивость* пары (*I*,*E*) для семейства функций *G* реализует, что если для программы *P*∈*G*, свойство *IP*(*x*1,...,*xk*)=0 удовлетворяется с высокой вероятностью, когда 〈*x*1,...,*xk*〉 выбрано с распределением *E* из *Dk*, тогда существует функция *g*∈*F*∩*G*, которая согласуется с *P* на большей части входов из *D*′.

Рассмотрим некоторое свойство линейности (*I*,*E*), где свойство *I f*(*x*1,*x*2,*x*3) тождественно *f*(*x*1)+*f*(*x*2)=*f*(*x*3) и *E* означает (*x*1∈R*Zp*,*x*2∈R*Zp*,*x*1+*x*2). Пара (*I*,*E*) характеризует *F*={*f*(*x*)=*cx*⏐*c*∈*Zp*} – множество всех линейных функций над *Zp*. В этом примере *G* – тривиальное множество всех функций и пара (*I*,*E*) устойчива для *G*.

Как только мы убедились посредством рандомизированных попыток, что программа *Р* удовлетворяет свойству устойчивости, можно переходить к тестированию программы на линейную и единичную состоятельность.

Существует два базовых теста для самотестирующихся программ – *тест линейной состоятельности* и *тест единичной состоятельности*. Продемонстрируем построение таких тестов на примере следующей тривиальной модулярной функции. Пусть *x*,*R* – положительные целые, тогда *fR*(*x*)≡*x* (mod *R*), где *R* - фиксировано.

Пусть *x*1 и *x*2 случайно, равновероятно и независимо от других событий выбраны из *ZR*2*n* и *x* принимает значение: *x*:≡*x*1+*x*2(mod *R*2*n*). Необходимо отметить, что *fR*(*x*)≡[*fR*(*x*1)+*fR*(*x*2)](mod *R*) – линейная функция по всем входам (аргументам). Тогда *тест линейной состоятельности* заключается в выполнении или не выполнении равенства: *PR*(*x*)≡[*PR*(*x*1)+*PR*(*x*2)](mod *R*), а *ошибка линейной состоятельности* – есть вероятность того, что данный тест не выполнился.

Пусть *z* случайно выбрано из *ZR*2*n* в соответствии с распределением *UZR*2*n* и *z* принимает значение: *z*′:≡*z*+1(mod*R*2*n*). Отметим также, что *fR*(*z*′)≡[*fR*(*z*)+1](mod *R*). Тогда *тест единичной состоятельности* заключается в выполнении или не выполнении равенства: *PR*(*z*′)≡[*PR*(*z*)+1](mod*R*), а *ошибка единичной состоятельности* – есть вероятность того, что данный тест не выполнился.

**Метод создания самокорректирующейся процедуры вычисления теоретико-числовой функции дискретного экспоненцирования**

Обозначения и определения для функции дискретного возведения в степень вида gxmodulo M.

Пусть *In*=*Zq* представляет собой множество {1,...,*q*}, где *q*=φ(*M*) - значение функции Эйлера для модуля *M*, а *Z\*M* - множество вычетов по модулю *M*, где *n*=⎡log2*M*⎤. Пусть распределение *Dp* есть равномерное распределение вероятностей. Оракульной программой, в данном случае, является программа вычисления функции *gx*modulo*M*, по отношению к исследуемым самотестирующейся и самокорректирующейся программам.

Пусть *x*[1,...,*n*] - двоичное представление положительного числа *x* и *A*, *B* и *N* - положительные целые числа в *r*-ичной системе счисления, тогда псевдокод алгоритма *Ax*modulo *N*, реализованного программой Exp() имеет следующий вид.

Program Exp(*x*,*N*,*A*,*R*); {вход *x*,*N*,*A*, выход *R*}

begin

*B*:=1;for *i*:=1 to *n* do

begin

*B*:=(*B*\**B*)mod*N*;

if [*i*]=1 then *B*:=(*A*\**B*)mod*N*;

end;

*R*:=*B*;

end.

Рис. 1. Псевдокод алгоритма *Ax*modulo*N*

Из описания алгоритма видно, что число обращений к операции вида (*A*\**B*)modulo*N* будет log*x*+β(*x*), где β(*x*) число единиц в двоичном представлении операнда *x* или вес Хэмминга *x*.

***Построение самотестирующейся/самокорректирующейся пр.ограммной пары для функции дискретного экспоненцирования***

Сначала рассмотрим следующие 4 алгоритма (ри.c2-5). Для доказательства полноты и безопасности указанной самотестирующейся/ самокорректирующейся программной пары доказывается следующая теорема.

Теорема 1. Пара программ S\_K\_exp(*x*,*M*,*q*,*g*,β) и S\_T\_exp(*x*,*M*,*q*,*g*,β) является (1/288,1/8,1/8)-самокорректирующейся/ самотестирующейся программной парой для функции *gx*modulo*M* с входными значениями, выбранными случайно и равновероятно из множества *In*.

Для доказательства теоремы необходимо доказать две леммы.

Лемма 1. Программа S\_K\_exp(*M*,*q*,*g*,β) является (1/8)-самокорректирующейся программой для вычисления функции *gx*modulo*M* в отношении равномерного распределения *Dn*.

Доказательство. Полнота программы S\_K() означает, что если оракульная программа *P*(*x*), обозначаемая как Exp() выполняется корректно, то и самокорректирующаяся программа S\_K() будет выполняться корректно. В данном случае полнота программы очевидна. Если *P*(*x*) корректно вычислима, то из [*PM,g*(*x*1) *PM.g*(*x*2)](mod*M*) следует, что

*fM,g*(*x*)=*fM,g*(*x*1)°*fM,g*(*x*2)=*g*[*x*1+*x*2](modφ(*M*))(mod*M*)≡*gx*(mod *M*)≡*Rk*.

Program S\_K\_exp(*x*,*M*,*q*,*g*, *Rk*); {вход *n*,*x*,*M*,*q*,*g*, выход *Rk*}

begin

for *l*=1 to 12 ln(1/β)

begin

*x*1:=random(*q*); {random- функция случайного равновероятного выбора из

отрезка вычисления целочисленного [1,...,*q*-1]}

*x*2:=(*x*-*x*1)mod*q*;

Exp(*g*,*x*1,*M*,*R*1); {Exp- процедура *gx*modulo*M=R*}

Exp(*g*,*x*2,*M*,*R*2);

*R*0:=(*R*1\**R*2)mod*M*;

end;*Rk*=choice(*R*0(*l*)); {choice- функция выбора из массива, состоящего из 12ln(1/β)

элементов, ответов, который повторяется

наибольшее количество раз}

end.

Рис. 2. Псевдокод алгоритма S\_K\_exp

Program S\_T\_exp(*x*,*M*,*q*,*g*,β); {вход *x*,*M*,*q*,*g*, выход значение предиката output}

Begin *t*1:=0;*t*2:=0;

for *l*=1 to 576 ln(4/β)

begin L\_T(*g,M,q*,*Rl*); {L\_T - процедура, реализующая тест линейной состоятельности, выход - *Rl*}

*t*1:=*t*1+*Rl*;

end;

if *t*1/576 ln(4/β)>1/72 then output:=«false»;

for *l*=1 to 32(4/β)

begin

N\_T(*g,M,q*,*Re*); {N-T - процедура, реализующая тест единичной состоятельности,

выход - *Re*}

*t*2:=*t*2+*R*;

end;

if *t*2/32 \*ln(4/β)>1/4 then output:= «false»

else output:=«true»

end.

Рис. 3. Псевдокод алгоритма S\_T\_exp

Program L\_T(*n*,*R*); {вход *g*,*M*,*q*, выход *Rl*}

begin

*x*1:=random(*q*);

*x*2:=random(*q*);

*x*:=(*x*1+*x*2)mod*q*;

Exp(*g*,*x*1,*M*,*R*1);

Exp(*g*,*x*2,*M*,*R*2);

Exp(*g*,*x*,*M*,*R*);

if *R*1\**R*2=*R* then Rl:=1

else Rl:=0;

end.

Рис. 4. Псевдокод алгоритма теста линейной состоятельности L\_T

Program N\_T(*n*,*R*); {вход *g*,*M*,*q*, выход *Re*}

begin

*x*1:=random(*q*);

*x*2:=(*x*1+1)mod*q*;

Exp(*g*,*x*1,*M*,*R*1);

Exp(*g*,*x*2,*M*,*R*2);

if *R1*\*g=*R*2 then *Re*:=1

else *Re*:=0;

end.

Рис. 5. Псевдокод алгоритма теста единичной состоятельности N\_T

Для доказательства безопасности сначала необходимо отметить, что так как *x*1∈R*Zq*, то и *x*2 имеет равномерное распределение вероятностей над *Zq*. Так как вероятность ошибки ε≤1/8, то в одном цикле вероятность Prob[*Rk*=*fM,g*(*x*)]≥3/4. Чтобы вероятность корректного ответа была не менее чем 1-β, исходя из оценки Чернова, необходимо выполнить не менее 12ln(1/β) циклов .

Лемма 2. Программа S\_T\_exp(*n*,*M*,*q*,*g*,β) является (1/288,1/8)- самотестирующейся программой, которая контролирует результат вычисления значения функции *gx*modulo*M* с любым модулем *M*.

Доказательство. Полнота программы S\_T() доказывается аналогично доказательству полноты в лемме 1, где *x*1,*x*2∈R*Zq*. Полнота теста единичной состоятельности вытекает исходя из следующего очевидного факта. Корректное выполнение теста [*PM,g*(*x*1)*PM.g*(1)](mod *M*) соответствует вычислению функции:

*fM,g*(*x*)=*fM,g*(*x*1)°*fM,g*(1)= *g* [ *x*1 +1](mod φ ( *M* )) ≡ *g x*1 *g* (mod *M* ) ≡*gx*(mod*M*)≡*Re*.

Для доказательства условия самотестируемости необходимо отметить, что так как и в лемме 1 для того, чтобы вероятность корректных ответов *Rl* и *Re* в обоих тестах была не более 1-β достаточно выполнить тест

линейной состоятельности 576 ln(4/β) раз и тест единичной состоятельности 32 ln(4/β) раз.

Можно показать, основываясь на теоретико-групповых рассуждениях, что возможно обобщение программы S\_T() и для других групп (вышеописанные алгоритмы основываются на вычислениях в мультипликативной группе вычетов над конечным полем). То есть для всех *y*∈*G*, *P*(*y*)∈*G\**, где *G\** -представляет собой любую группу, кроме групп *G\*\**. К группам последнего вида относятся бесконечные группы, не имеющие конечных подгрупп за исключением {*О*′}, где *О*′ - тождество группы. Таким образом, можно показать (если параметры выбираются независимо, равновероятно и случайным образом), что программа вида S\_T() является (ε/36,ε)-самотестирующейся программой. Из всего сказанного, следует доказательство утверждения леммы .

Исходя из определения самотестирующейся/ самокорректирующейся программной пары и основываясь на результатах доказательств лемм 1 и 2, очевидным образом следует доказательство теоремы 1 .

*Замечания*. Как видно из псевдокода алгоритма *Ax*modulo *N* в нем используется операция *AB*modulo*N*. Используя ту же технику доказательств, как и для функции дискретного возведения в степень, можно построить (1/576,1/36,1/36)-самокорректирующуюся/ самотестирующуюся программную пару для вычисления функции модулярного умножения. Это справедливо, исходя из следующих соображений. Вычисление функции *fM*(*ab*)=*fM*((*a*1+*a*2)(*b*1+*b*2)) следует из корректного выполнения программы с 4-х кратным вызовом оракульной программы *P*(*a*,*b*), то есть:

[*PM*(*a*1,*b*1)+*PM*(*a*1,*b*2)+*PM*(*a*2,*b*1)+*PM*(*a*2,*b*2)](mod *M*).

Алгоритм вычисления *Ax*modulo*N* выполняется для *c*=2. Однако, декомпозиция *x*, как следует из свойства самосводимости функции *Ax*modulo *N*, может осуществляться на большее число слагаемых. Хотя это приведет к гораздо большему количеству вызовов оракульной программы, но в то же время позволит значительно снизить вероятность ошибки.

**Метод создания самотестирующейся расчетной программы с эффективным тестирующим модулем**

В качестве расчетной программы рассматривается любая программа, решающая задачу получения значения некоторой вычислимой функции.

При этом под верификацией расчетной программы понимается процесс доказательства того, что программа будет получать на некотором входе истинные значения исследуемой функции.

Пусть для функции *Y = f* (*X*) существует пара функций (*gc*, *hc*)*Y* таких, что:

*Y* = *gc* (*f* (*a*1), ..., *f* (*ac*)),

*X* = *hc* (*a1*, ..., *ac*).

Легко увидеть, что если значения *ai* выбраны из *In* в соответствии с распределением *Dp*, тогда пара функций (*gc*, *hc*)Y обеспечивает выполнение для функции *Y = f* (*X*) свойства случайной самосводимости. Пару функций (*gc*, *hc*)Y будем называть *ST-парой функций* для функции *Y = f* (*X*).

Предположим, что на *ST*-пару функций можно наложить некоторую совокупность ограничений на сложность программной реализации и время выполнения. В этом случае, пусть длина кода программ, реализующих функции *gc* и *hc* , и время их выполнения составляет константный мультипликативный фактор от длины кода и времени выполнения программы *P*.

Предлагаемый метод верификации расчетной программы *P* на основе *ST*-пары функций для некоторого входного значения вектора *X*\* заключается в выполнении следующего алгоритма. (Всюду далее, если осуществляется случайный выбор значений, этот выбор выполняется в соответствии с распределением вероятностей *Dp*).

**Алгоритм ST**

1. Определить множество *A\**= {*a1\****, . . . ,** *ac\* }* такое, что *X***\*** = *h{a1\****, . . . ,** *ac\* }*,

где *a1\****, . . . ,** *ac\** выбраны случайно из входного подмножества *In*.

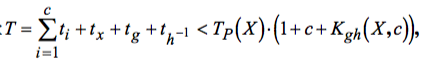
2. Вызвать программу *P* для вычисления значения *Y0\**=*f(X\*)***.**

3. Вызвать *c* раз программу *P* для вычисления множества значений {*f (a1\*)* **,...,***f (a1\* )*}

4. Определить значения *Y1* = *gc{ ( a1\* )***,...,** *f (ac\*)}* **.**

5. Если *Y0***\*** = *Y1***\*** , то принимается решение, что программа *P*  корректна на множестве значений входных параметров {*X***,***a1\****,...,***ac\**}**,** в противном случае данная программа является некорректной.

Таким образом, данный метод не требует вычисления эталонных значений и за одну итерацию позволяет верифицировать корректность программы *P* на (*n*+1) значении входных параметров. При этом время верификации можно оценить

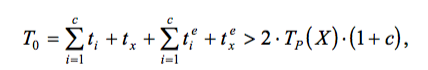
как 

где *ti* и *tx* - время выполнения программы *P* при входных значениях *ai* и *X\** соответственно;

*tg* и *th-1*- время определения значения функции *gc* и множества *A\** соответственно:

*TP* (*X*) - временная (не асимптотическая) сложность выполнения программы *P*;

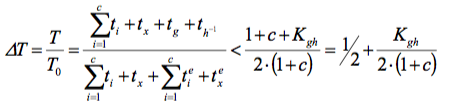
*Kgh* (*X*, *c*) - коэффициент временной сложности программной реализации функции *gc* и определения *A\** по отношению к временной сложности программы *P* (по предположению он составляет константный мультипликативный фактор от *TP*(*X*), а его значение меньше 1). Для традиционного вышеуказанного метода тестирования время выполнения и сравнения полученного результата с эталонным значением составляет:

**

 где *tie* и *txe* - время определения эталонных значений функции

*Y* = *f* (*X*) при значениях *ai* и *X\** соответственно (в общем случае, не может быть меньше времени выполнения программы).

Следовательно, относительный выигрыш по оперативности предложенного метода верификации (по отношению к методу тестирования программ на основе ее эталонных значений):

**

Так как, коэффициент *Kgh* < 1, а *c* ≥ 2, то получаем относительный выигрыш по оперативности испытания расчетных программ указанного типа (обладающих свойством случайной самосводимости) более чем в 1.5 раза.

**Исследования процесса верификации расчетных программ**

В качестве примера работоспособности предложенного метода рассмотрим верификацию программы вычисления функции дискретного возведения в степень:

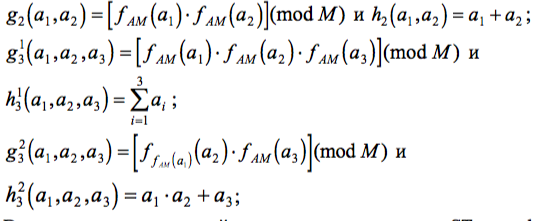
*y*=*fAM* (*x*)=*Ax*modulo*M*.

Выбор данной функции обусловлен тем, что она является одной из основных функций в различных теоретико-числовых конструкциях, например, в схемах электронной цифровой подписи, системах открытого распределения ключей и т.п. Это, в свою очередь, демонстрирует возможность применения предложенного метода при исследовании расчетных программ, решающих конкретные прикладные задачи. Кроме того, очевидно, что данная функция обладает свойством случайной самосводимости, а, исходя из вышеприведенных рассуждений можно показать, что для данной функции существует (1/288, 1/8)-самотестирующаяся программа.

Для экспериментальных исследований была выбрана программа EXP из библиотеки базовых криптографических функций CRYPTOOLS [КС2], которая реализует функцию дискретного экспоненцирования в степень (размерность переменных и констант не превышает 64 или 128 байтов).

Была разработана интегрированная оболочка для проведения верификации, включающая интерфейс с пользователем и программные процедуры, реализующие некоторую совокупность проверочных тестов. Экспериментальные исследования состояли из определения временных характеристик процесса верификации на основе использования *ST*-пары функций и определения возможности обнаружения предложенным методом преднамеренно внесенных программных ошибок.

Для этого были определены следующие *ST*-пары функций:



В процессе исследований менялась используемая *ST*-пара функций и варьировалась размерность параметров *A*, *M* и аргумента *X*. Результаты экспериментов полностью подтвердили приведенные выше временные зависимости (технические результаты исследований автор в данной работе опускает).

Исследование возможности обнаружения предложенным методом преднамеренно внесенных закладок заключалось в написании программы EXPZ. Спецификация для программ EXP и EXPZ одна и та же, отличие же заключается в том, что программа EXPZ содержит программную закладку деструктивного характера. Преднамеренно внесенная закладка при исследованиях гарантировала сбой работы программы вычисления значения функции *y* = *fAM* (*x*) = *Ax* modulo *M* (то есть обеспечивала получение неправильного значения функции) примерно на каждой восьмой части входных значений экспоненты *x*.

Было проведено свыше 2000 экспериментов[КС2]. Все входные значения, на которых произошел сбой программы, были обнаружены, что в дальнейшем подтвердилось проверочными тестами, основанными на использовании малой теоремы Ферма и теореме Эйлера. Этот факт, в свою очередь, экспериментально показал, что программа, реализующая алгоритм **ST**, является (1/8,1/288)-самотестирующейся программой.

Таким образом, предложенный метод позволяет в значительной степени сократить время испытания расчетных программ на предмет выявления непреднамеренных и преднамеренных программных дефектов. При этом по результатам испытаний можно получить количественные оценки вероятности наличия программных дефектов в верифицируемой расчетной программе. Однако, разработка формальных методов получения *ST*-пары функций для заданной расчетной программы, а также разработка методик ее испытания с помощью рассмотренного алгоритма требуют дальнейших теоретических и прикладных исследований.

**Области применения самотестирующихся и самокорректирующихся программ и их сочетаний**

Применение чекеров, самотестирующихся, самокорректирующихся программ и их сочетаний возможно в самых различных областях вычислительной математики, а, следовательно, в самых разнообразных областях человеческой деятельности, где широко применяются вычислительные методы. К ним относятся такие направления как цифровая обработка сигналов (а, следовательно, решение проблем в системах распознавания изображений, голоса, в радио- и гидроакустике), а также методы математического моделирования процессов изменения народонаселения, экономических процессов, процессов изменения погоды и т.п. Идеи самотестирования могут найти самое широкое применение в системах защиты информации, например, в системах открытого распределения ключей, в криптосистемах с открытым ключом, в схемах идентификации пользователей вычислительных систем и аутентификации данных, где базовые вычислительные алгоритмы обладают некоторыми алгоритмическими свойствами, например, свойством случайной самосводимости, описанным выше.

***Вычислительная математика***

Активными исследованиями в области создания самотестирующихся и самокорректирующихся программ ученые и практики начали заниматься с начала 90-х годов. В этот период были разработаны программные чекеры для ряда теоретико-числовых и теоретико-групповых задач, для решения задач с матрицами, полиномами, линейными уравнениями и рекуррентными соотношениями.

*Теоретико-групповые и теоретико-числовые вычисления*

*Проблема эквивалентного поиска.* Пусть *S* – множество и *G* – группа, групповые действия в которой, осуществляются над элементами множества *S*. Для *a*,*b*∈*S* элемент *a*≡*Gb*, тогда и только тогда, когда *g*(*a*)=*b* для некоторого *g* из *G*. Проблема эквивалентного поиска заключается в нахождении *g* такого, что *g*(*a*)=*b*, если *a*≡*Gb* для *a* и *b*, принадлежащих множеству *S*. Если существует эффективный вероятностный алгоритм поиска *g*∈*G*, тогда существует [BK] эффективный программный чекер для данной проблемы. Примеров решения задач эффективного эквивалентного поиска достаточно много. К ним относятся проблема поиска изоморфизма графов (см. дальше), решение задачи квадратных вычетов, обобщенная проблема дискретных логарифмов, задачи подобные «Кубику Рубика», ряд задач из теории кодирования и др.

*Проблема пересечения групп.*

Пусть *G* и *H* – группы перестановок, определенные некоторыми генераторами групп. Генераторы представляются как перестановки над [1,...,*n*]. Проблема пересечения групп заключается в нахождении генераторов для *G*∩*H*.

*Проблема расширенного нахождения НОД.* Проблема расширенного нахождения наибольшего общего делителя (которая отличается от нахождения НОД посредством алгоритма Евклида) заключается в нахождении для двух положительных целых *a* и *b* целого *d*=НОД(*a*,*b*) и целых *u* и *v* таких, что *au*+*bv*=*d*.

Чекер для решения расширенного нахождения НОД по входу двух положительных целых *a* и *b*, целого *d* и целых *u* и *v* выдать «Сбой», если *d* не делит *a* или *d* не делит *b* или *au*+*bvd*. В противном случае выдать «Норма». Эффективность и корректность данного чекера легко доказывается.

*Вычисления над полиномами*

Существует достаточно простой способ построения самокорректирующейся программы, который основывается на существовании следующего интерполяционного тождества, соответствующего значения функций между точками: для всех одномерных полиномов *f* степени не более*d*, для всех *x*,*t*∈*F*, *d*+1 , где *ai* – различные элементы из *F*, *ai* =-1 и *ai* зависит от *F*,*d* и не зависит от *x*,*t*. Тогда самокорректирующаяся программа для вычисления *f(*)=*f*(*x*1,...,*xn*) заключается в выполнении следующего алгоритма. Случайно и равновероятно выбирается *t* =(,...,*tn*) и выдается .С вероятностью не менее 2/3 все вызовы программы *i*=0 будут возвращать корректные результаты и, следовательно, выход программы будет корректным. Рис.6 демонстрирует самотестирующуюся программу для полинома *f*.

Program P\_S\_T(*P*,ε,β,*x*,*f*(*x*)); {вход *P*,ε,β,(*x*1,*f*(*x*1)),...,*xd*+1,*f*(*xd*+1))),

выход («Норма»,«Сбой»)}

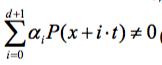
begin

for *i*=1 to O((1/*k*)log(1/β)) do

begin

*x*,*t*:=random(*Zp*); {random - функция случайного равновероятного выбора

из множества вычетов по модулю *р*};

if  (более, чем в ε итерациях) then

output «Сбой»;

end;

output «Норма»;

for *j*=0 to *d* do

begin

for *i*=1 to O(log(*d*/β)) do

begin

*t*:=random(*Zp*); {random - функция случайного равновероятного выбора

из множества вычетов по модулю р};

*if * (более, чем в 1⁄4 *i*=0

итерациях) then output «Сбой»;

end;

end;

output «Норма»;

end.

Рис.6. Псевдокод алгоритма самотестирующейся программы для полинома *f*

В ряде работ было показано, как строить самотестирующиеся и самокорректирующиеся программы для задач сложения и умножения полиномов и аппроксимирующие чекеры для полиномов.

*Вычисления над матрицами*

Пусть матрицы *А* и *В* матрицы размером *n*×*n* определены над конечным полем *F*.

Время выполнения программы, реализующей чекер Фрейвалдса, составляет *O*(*n*2log(1/*k*)).

Используя чекер Фрейвалдса, можно построить самотестирующуюся/ самокорректирующуюся программную пару для умножения матриц (рис.8 – 9).

Program C\_F(*A,B,C*,*k*); {вход *A,B,C*,*k*, выход («Норма»,«Сбой»)}

begin

for *i*=1 to log(1/*k*) do begin

*R*:=random(*F*); {random - функция случайного равновероятного выбора

0-1- вектора размером (*n*×1) из *F*};

if *CRA*(*BR*) then output «Сбой»;

end;

output «Норма»;

end.

Рис.7. Псевдокод алгоритма, реализующего чекер Фрейвалдса

Program S\_K\_multAB(*A,B,k*); {вход *A,B*,*k*, выход *С*}

begin

for *i*=1 to ∞ do

begin

*A*1:=random(*F*); {random - функция случайного равновероятного выбора

матрицы размером (*n*×*n*) из *F*};

*B*1:=random(*F*); {random - функция случайного равновероятного выбора

матрицы размером (*n*×*n*) из *F*};

*A*2:=*A*-*A*1;

*B*2:=*B-B*1;

*C*:=*P(A*1,*B*1)+*P*(*A*1,*B*2)+*P*(*A*2,*B*1)+*P*(*A*2,*B*2);

 if C\_F(*A,B,C*,*k*)=«Норма» then output *С* and goto 1;

end;

1:end.

Рис.8. Псевдокод алгоритма самокорректирующейся программы умножения матриц

Время выполнения программы S\_K\_multAB составляет *О*(*M*(*n*)+*n*2log(1/*k*)), где *M*(*n*) – время выполнения программы умножения матриц размером *n*×*n* .

Самотестирующаяся программа для умножения матриц строится достаточно просто.

Program S\_T\_multAB(*A,B,k*); {вход *A,B*,*k*, выход («Норма»,«Сбой»)}

begin

for *i*=1 to O(log(1/k))∞ do

begin

*A*:=random(*F*); {random - функция случайного равновероятного и независимого

выбора матрицы размером (*n*×*n*) из *F*};

*B*:=random(*F*); {random - функция случайного равновероятного и независимого

выбора матрицы размером (*n*×*n*) из *F*};

*C*:=*P(A*,*B*);

if C\_F(*A,B,C*,1/4)=«Норма» then output 0 and goto 1;

if C\_F(*A,B,C*,1/4)=«Сбой» then output 1 and goto 1;

end;

1:end.

Рис. 9. Псевдокод алгоритма самотестирующейся программы умножения матриц

Можно легко удостовериться, что, если *err*(*P*,*f*,*Un*)≥1/8, то количество единиц будет не менее 1/16 с вероятностью не менее 1-*k* и если *err*(*P*,*f*,*Un*)≤1/32, то количество единиц будет не менее 1/16 с вероятностью не менее 1-*k*. Таким образом, вышеприведенная программа будет (1/32,1/8)-самотестирующейся программой для умножения матриц.

*Линейные рекуррентные соотношения*

В работе [KS] исследовались вопросы построения самотестирующихся и самокорректирующихся программ для линейныхрекуррентных соотношений, т.е. соотношений вида  .

Такие последовательности являются основными для многих комбинаторных и теоретико-числовых последовательностей, таких как последовательность Фибоначчи и последовательность Лукаша.

Линейные рекуррентные соотношения часто рассматриваются в неявном виде в качестве однородных линейных уравнений вида:

. Линейные рекуррентные соотношения часто используются в таких прикладных областях как моделирование динамики изменения народонаселения, различных экономических процессах, при анализе различных трафиков (потоков всевозможных данных, процессов) и т.п. Кроме того, такие последовательности используются при описании различных процессов в робототехнике и цифровой обработке сигналов.

*Аппроксимирующие функции*

Пусть для данной функции *f* и границы ошибки δ, входа *x*, программа

*P*(*x*) вычисляющая функцию *f*, *приблизительно корректна*, если |*P*(*x*)-*f*(*x*)|≤δ. Это обозначается следующим образом: *P*(*x*)≈δ*f*(*x*). Будем также считать, что *Р*(δ,ε) *аппроксимирует функцию f* на области *D*, если |*P*-*f*|≤δ на 1-ε элементах области *D*.

***Криптография, интерактивные доказательства***

*Вводные замечания*

Основная идея использования задач самотестирования в криптографии заключается в девизе «Защитить самих защитников». Так как криптографические методы используются для высокоуровневого обеспечения конфиденциальности и целостности информации, собственно программно-техническая реализация этих методов должна быть свободна от программных и/или аппаратных дефектов. Таким образом, самотестирование и самокоррекция программ может эффективно применяться в современных системах защиты информации от несанкционированного доступа.

*Распределение ключей, цифровая подпись, схемы аутентификации*

Функция дискретного экспоненцирования, описанная выше, широко используется в современной криптографии, в частности, при открытом распределении ключей Диффи-Хеллмана, для генерации и верификации подписей в схемах электронной цифровой подписи, для построения различных схем аутентификации сообщений, идентификации пользователей вычислительных систем и т.п. Следовательно, существует принципиальная возможность построения программных чекеров, самотестирующихся, самокорректирующихся программ. Продемонстрируем это на примере схемы цифровой подписи*RSA* (рис. 10).

Пусть программа *P* предположительно вычисляет *RSA*-функцию и для *x*,*y*,*z*∈*Z*>0 с *x*,*y*<*z* и НОД(*x*,*z*)=1. Тогда чекер *CPRSA*(*x*,*y*,*z*;*k*) работает следующим образом [KA].

Доказательства существования чекера для *RSA*-криптоалгоритма приведены в работе[KA]. Там же приведены *RSA*-чекеры для фиксированного модуля криптосистемы.

Program C\_RSA\_(*x*,*y*,*z*;*k*); {вход *x*,*y*,*z*,*k* выход («Норма»,«Сбой»)}

begin

*t*1:=-*k*log99/1002;

*t*2:=-*k*/log4/52;

for *l*=1 to *t*1 do

begin

*i*:=random(*Z*); {random - функция случайного равновероятного выбора из [1,...,*z*[};

if *P*(*x*,*i*,*z*)≡0 (mod *z*) output «Сбой» and STOP;

*i*,*j*:=random(*Z*); {random - функция случайного равновероятного выбора из [1,...,*z*[};

if *P*(*x*,*i*,*z*)*P*(*x*,*j*,*z*)*P*(*x*,*i*+*j*,*z*)(mod *z*) output «Сбой» and STOP;

*i*:=random(*Z*); {random - функция случайного равновероятного выбора из [1,...,*z*[};

if *P*(*x*,*i*,*z*)≡*P*(*x*,1,*z*)*P*(*x*,*i*+1,*z*)(mod*z*) output «Сбой» and STOP;

end;

for *l*=1 to *t*2 do

begin *r*:=random(*Z*); {random - функция случайного равновероятного выбора из [1,...,z[};

if *P*(*x*,*y*,*z*)*P*(*x*,*r*,*z*)≠*P*(*x*,*y*+*r*,*z*)(mod *z*) «Сбой» and STOP;

end;

output «Норма»;

output

end.

Рис.10. Псевдокод алгоритма *RSA*-чекера

*Интерактивные системы доказательств*

Пусть **А** и **В** *-* две интерактивные, т. е. взаимодействующие через общую коммуникационную ленту, вероятностные машины Тьюринга. Через [*В*(*х*)*,А*(*х*)] обозначается случайная величина - выходное слово машины **А***,* когда **А** и **В** работают на входном слове*х*. Через |*х*|

обозначается длина слова *х*. *Интерактивным доказательством для языка L* называется пара интерактивных машин Тьюринга (**P**,**V**) такая, что выполняются следующие два условия.

1. (*Полнота*). Для всех *х*∈*L* вероятность Prob{[*P*(*x*),*V*(*x*)]=1}=1.

2. (*Корректность*). Для любой машины Тьюринга **Р*\****, для любого полинома *р* и для всех *х*∈*L* достаточно большой длины Prob{[*P\**P (*x*),*V*(*x*)]=1}<l/*p*(|*x*|).

Формальные определения интерактивных систем доказательств и интерактивных систем доказательств с нулевым разглашением даны в приложении.

*Задача «Изоморфизм графа»*

Приведем в качестве примера протокол доказательства с абсолютно нулевым разглашением для языка «*Изоморфизм графов*». Входным словом является пара графов *G*1=(*U,Е*1) и *G*0=(*U,E*0)*.* Здесь *U* - множество вершин, которое можно отождествить с множеством натуральных чисел {1,...,*n*}, *Е*1 и *Е*0 множества ребер такие, что |*Е*1|=|*Е*0|=*m.* Графы *G*1 и *G*2 называются изоморфными, если существует перестановка на множестве *U* такая, что (*u*,*v)*∈*Е*0 тогда и только тогда, когда (φ(*u*)*,*φ(*v*))∈*Е*1 (обозначается *G*1=φ*G*0). Задача распознавания изоморфизма графов хорошо известная математическая задача, для которой на данный момент неизвестно полиномиальных алгоритмов ее решения. Сдругой стороны, неизвестно, является ли эта задача **NР**- полной, хотя есть веские основания предполагать, что не является.

**Протокол IG**

Пусть φ изоморфизм между *G*1 и *G*0. Следующие четыре шага выполняются в цикле *m* раз, каждый раз с независимыми случайными величинами.

1. **Р** выбирает случайную перестановку π на множестве *U,* вычисляет *Н*=-*тG*1 и посылает этот граф **V.**

2. **V** выбирает случайный бит α и посылает его **Р**.

3. Если α=1, то **Р** посылает **V** перестановку π, в противном

случае - перестановку π°φ*.*

4. Если перестановка, полученная **V**, не является изоморфизмом между *G*α и *Н,* то **V** останавливается и отвергает доказательство. В противном случае выполнение протокола продолжается.

Если проверки п.4 дали положительный результат во всех *m* циклах, то **V** принимает доказательство.

Необходимо отметить, что если в протоколе **IG** машина **Р** получает изоморфизм в качестве дополнительного входного слова, то ей для выполнения протокола не требуются неограниченные вычислительные ресурсы. Более того, в этом случае **Р** может быть полиномиальной вероятностной шиной Тьюринга.

Теорема1. Протокол **IG** является доказательством с абсолютно нулевым разглашением для языка «Изоморфизм графов»*.*

Приводим без доказательств. Доказательство можно найти в литературе.

*Чекер для задачи «Изоморфизм графа»*

Пусть *G* и *H* – два графа и *k* – некоторый параметр безопасности.

Чекер *CGIP*(*G*,*H*,*k*) проверяет программу *P* на входных графах *G* и *H*. На выходе чекера будет результат «Норма», если графы изоморфны и «Сбой», если неизоморфны.

Следующая теорема [BK] определяет эффективность и корректность чекера для решения задачи «ИЗОМОРФИЗМ ГРАФОВ» - **IG**.

Пусть *P* – программа, которая останавливается на всех входах и всегда выдает либо «Норма», либо «Сбой». Пусть также *G* и *H* - два любых графа и пусть *CGIP*(*G*,*H*;*k*) – вышеуказанный чекер.

Теорема 2*.* Если *P* корректная программа для решения задачи **IG**, тогда чекер *CGIP*(*G*,*H*;*k*) всегда выдаст на выходе «Норма». Если *P* - некорректна, т.е. *P*(*G*,*H*)*GI*(*G*,*H*), тогда вероятность Prob{*CGIP*(*G*,*H*;*k*)=«Норма»}<1/2*k*.

Приводим без доказательств. Доказательство можно найти в литературе.

Интерактивные системы доказательств и интерактивные системы доказательств с нулевым разглашением могут эффективно применяться в системах защиты информации от несанкционированного доступа, например, в схемах интерактивной идентификации пользователей системы. В целом интерактивный протокол доказательств для задачи **IG** может применяться для идентификации, однако для практических целей удобно использовать системы доказательств, основанные на трудности решения некоторых теоретико-числовых задач. Тем более, как было показано выше, существует принципиальная возможность построения самотестирующейся/ самокорректирующейся программной пары, например, для функции дискретного экспоненцирования.

**Чекер *CGIP*(*G*,*H*,*k*)**

1. Вычислить *P*(*G*,*H*).

2.Если *P*(*G*,*H*)=«Изоморфизм», тогда использовать *P* для поиска изоморфизма их *G* в *H*. Проверить, является ли результирующее соответствие изоморфизмом. Если нет, тогда подать на выход «Сбой», если является, тогда подать на выход «Норма».

3. Если *P*(*G*,*H*)=«Не изоморфизм», тогда выполнить следующие шаги *k* раз.

3.1. Выработать случайный бит *b*.

3.2. Если *b*=1, тогда

3.2.1. Сгенерировать случайную перестановку *G*′ графа *G*.

3.2.2. Вычислить *P*(*G*,*G*′).

3.2.3. Если *P*(*G*,*G*′)=«Не изоморфизм», тогда подать на выход «Сбой».

3.3. Если *b*=0, тогда

3.3.1. Сгенерировать случайную перестановку *H*′ графа *H*.

3.3.2 Вычислить *P*(*G*,*H*′).

3.4. Если *P*(*G*,*H*′)=«Изоморфизм», тогда подать на выход «Сбой».

4. Подать на выход «Норма».