Оглавление

[Введение 4](#_Toc390064754)

[1. Постановка задачи 5](#_Toc390064755)

[2. Аналитическая часть 6](#_Toc390064756)

[2.1. Система шифрования RSA 7](#_Toc390064757)

[2.2. Сложность теоретико-числовых алгоритмов 11](#_Toc390064758)

[2.2.1. Алгоритм вычисления  12](#_Toc390064759)

[2.2.2. Алгоритм Евклида 12](#_Toc390064760)

[2.2.3. Алгоритм решения уравнения  13](#_Toc390064761)

[2.2.4. Алгоритм нахождения делителей многочлена  в кольце  15](#_Toc390064762)

[2.2. Качественная теория алгоритма RSA 18](#_Toc390064763)

[2.2.1. Алгоритм доказывающий не простоту числа 19](#_Toc390064764)

[2.2.2. Нахождение больших простых чисел 21](#_Toc390064765)

[2.2.3. Проверка большого числа на простоту 24](#_Toc390064766)

[Выводы по главе 30](#_Toc390064767)

[3. Проектирование 31](#_Toc390064768)

[3.1. Проектирование функций 31](#_Toc390064769)

[3.2. Проектирование классов 33](#_Toc390064770)

[3.3. Проектирование классов исключений 35](#_Toc390064771)

[3.4. Модуль rsa.bigfile 36](#_Toc390064772)

[3.6. Модуль rsa.core 37](#_Toc390064773)

[Выводы по главе 37](#_Toc390064774)

[4. Тестирование и документирование 38](#_Toc390064775)

[4.1. Тестирование 38](#_Toc390064776)

[4.2. Руководство программиста 39](#_Toc390064777)

[4.2.1. Назначение и область применения 39](#_Toc390064778)

[4.2.2. Характеристики программы 39](#_Toc390064779)

[4.2.3. Обращение к программе 40](#_Toc390064780)

[4.2.3.1. Генерация ключа 40](#_Toc390064781)

[4.2.3.3. Требования к размеру ключа 42](#_Toc390064782)

[4.2.3.4. Шифрование и дешифрование 42](#_Toc390064783)

[4.2.3.5. Низкоуровневые операции 44](#_Toc390064784)

[4.2.3.6. Подписывание и проверка 44](#_Toc390064785)

[4.2.3.7. Работа с большими файлами 45](#_Toc390064786)

[4.2.3.7.1. Применение 46](#_Toc390064787)

[4.2.3.7.2. Использование Python-RSA: VARBLOCK формат 46](#_Toc390064788)

[4.2.3.7.3. Формат VARBLOCK 48](#_Toc390064789)

[4.2.3.7.3.1. Простое сообщение 48](#_Toc390064790)

[4.2.3.7.3.2. Варианты формата base 128 48](#_Toc390064791)

[4.2.3.7.3.3. Структура сообщений 49](#_Toc390064792)

[4.3. Руководство пользователя 51](#_Toc390064793)

[4.3.1. Общие сведения о программе 51](#_Toc390064794)

[4.3.2. Структура системы 52](#_Toc390064795)

[4.4. Руководство администратора 52](#_Toc390064796)

[Вывод по главе 53](#_Toc390064797)

[5. Экономическое обоснование разработки 54](#_Toc390064798)

[5.1. Описание работы 54](#_Toc390064799)

[5.2. Описание конкурентов 54](#_Toc390064800)

[5.3. Описание рынка сбыта 54](#_Toc390064801)

[5.4.1. Расчет трудоемкости 54](#_Toc390064802)

[5.4.2. Определение цены программной продукции 60](#_Toc390064803)

[5.4.3. Капитальные вложения 65](#_Toc390064804)

[5.4.4. Эксплуатационные расходы 66](#_Toc390064805)

[5.4.5. Показатель эффективности и годового экономического эффекта 66](#_Toc390064806)

[Вывод по главе 68](#_Toc390064807)

[Вывод по главе 69](#_Toc390064808)

[Заключение 70](#_Toc390064809)

[Список литературы 71](#_Toc390064810)

Введение

1. Постановка задачи
   1. Формулировка задачи

Безопасность передачи данных по каналам связи является актуальной. Современные компьютерные сети не исключение. К сожалению, в сетевых операционных системах (Windows NT/XP/Server, Novell и т.д.) иностранного производства, как следствие, из-за экспортных соображений уровень алгоритмов шифрования заметно снижен.

Исследовать современные методы шифрования и их применимость к шифрованию потоков данных. Разработать собственную библиотеку алгоритмов шифрования и программный продукт, демонстрирующий работу этих алгоритмов при передаче данных в сети.

* 1. Выводы

1. Аналитическая часть

Труды Евклида и Диофанта, Ферма и Эйлера, Гаусса, Чебышева и Эрмита содер­жат остроумные и весьма эффективные алгоритмы решения диофантовых уравнений, выяснения разрешимости сравнений, построения больших по тем временам простых чисел, нахождения наилучших приближений и т.д. В последние два десятилетия, благодаря в первую очередь запросам криптографии и широкому распространению ЭВМ, исследова­ния по алгоритмическим вопросам теории чисел переживают период бур­ного и весьма плодотворного развития.

Вычислительные машины и электронные средства связи проникли практически во все сферы человеческой деятельности. Немыслима без них и современная криптография. Шифрование и дешифрование текстов можно представлять себе как процессы переработки целых чисел при помощи ЭВМ, а способы, которыми выполняются эти операции, как неко­торые функции, определённые на множестве целых чисел. Всё это делает естественным появление в криптографии методов теории чисел. Кроме того, стойкость ряда современных криптосистем обосновывается только сложностью некоторых теоретико-числовых задач.

Но возможности ЭВМ имеют определённые границы. Приходится раз­бивать длинную цифровую последовательность на блоки ограниченной длины и шифровать каждый такой блок отдельно. Мы будем считать в дальнейшем, что все шифруемые целые числа неотрицательны и по вели­чине меньше некоторого заданного (скажем, техническими ограничени­ями) числа m. Таким же условиям будут удовлетворять и числа, получае­мые в процессе шифрования. Это позволяет считать и те, и другие числа элементами кольца вычетов . Шифрующая функция при этом может рассматриваться как взаимнооднозначное отображение колец вычетов



а число  представляет собой сообщение  в зашифрованном виде.

Простейший шифр такого рода - шифр замены, соответству­ет отображению  при некотором фиксированном целом k. Подобный шифр использовал еще Юлий Цезарь. Конечно, не каждое отображение  подходит для целей надежного сокрытия инфор­мации.

В 1978 г. американцы Р. Ривест, А. Шамир и Л. Адлеман (R.L.Rivest. A.Shamir. L.Adleman) предложили пример функции , обла­дающей рядом замечательных достоинств. На её основе была построена реально используемая система шифрования, получившая название по пер­вым буквам имен авторов -система RSA. Эта функция такова, что

1) существует достаточно быстрый алгоритм вычисления значений ;

2) существует достаточно быстрый алгоритм вычисления значений об­ратной функции ;

3) функция  обладает некоторым «секретом», знание которого позво­ляет быстро вычислять значения ; в противном же случае вычисле­ние  становится трудно разрешимой в вычислительном отношении задачей, требующей для своего решения столь много времени, что по его  
прошествии зашифрованная информация перестает представлять инте­рес для лиц, использующих отображение  в качестве шифра.

Еще до выхода из печати статьи копия доклада в Массачусетском Технологическом институте, посвящённого системе RSA. была послана известному популяризатору математики М. Гарднеру, который в 1977 г. в журнале Scientific American опубликовал статью посвящённую этой системе шифрования. В русском переводе заглавие статьи Гарднера зву­чит так: Новый вид шифра, на расшифровку которого потребуются мил­лионы лет. Именно эта статья сыграла важнейшую роль в распростране­нии информации об RSA, привлекла к криптографии внимание широких кругов неспециалистов и фактически способствовала бурному прогрессу этой области, произошедшему в последовавшие 20 лет.

* 1. Система шифрования RSA

Пусть  и  натуральные числа. Функция  реализующая схему RSA, устроена следующим образом

. (1)

Для расшифровки сообщения  достаточно решить сравнение

. (2)

При некоторых условиях на  и  это сравнение имеет единственное решение .

Для того, чтобы описать эти условия и объяснить, как можно найти решение, нам потребуется одна теоретико-числовая функция, так назы­ваемая функция Эйлера. Эта функция натурального аргумента  обозна­чается  и равняется количеству целых чисел на отрезке от 1 до , взаимно простых с . Так  и  для любого простого числа  и натурального . Кроме того,  для лю­бых натуральных взаимно простых  и . Эти свойства позволяют легко вычислить значение , если известно разложение числа  на простые сомножители.

Если показатель степени  в сравнении (2) взаимно прост с , то сравнение (2) имеет единственное решение. Для того, чтобы найти его, определим целое число , удовлетворяющее условиям

. (3)

Такое число существует, поскольку , и притом единствен­но. Здесь и далее символом  будет обозначаться наибольший общий делитель чисел  и . Классическая теорема Эйлера, утверждает, что для каждого числа , взаимно простого с , выполняется сравнение  и, следовательно.

. (4)

Таким образом, в предположении , единственное решение срав­нения (2) может быть найдено в виде

. (5)

Если дополнительно предположить, что число  состоит из различных простых сомножителей, то сравнение (5) будет выполняться и без предпо­ложения . Действительно, обозначим  и . Тогда  делится на , а из (2) следует, что . Подобно (4), теперь легко находим . А кроме того, имеем . Получившиеся сравнения в силу  дают нам (5).

Функция (1), принятая в системе RSA, может быть вычислена доста­точно быстро. Обратная к  функция  вычисляется по тем же правилам, что и , лишь с заменой показателя степени  на . Таким образом, для функции (1) будут выполнены указанные выше свойства 1) и 2).

Для вычисления функции (1) достаточно знать лишь числа  и . Именно они составляют открытый ключ для шифрования. А вот для вы­числения обратной функции требуется знать число . оно и является «се­кретом», о котором речь идёт в пункте в). Казалось бы. ничего не стоит. зная число . разложить его на простые сомножители, вычислить затем с помощью известных правил значение  и, наконец, с помощью (3) определить нужное число . Все шаги этого вычисления могут быть реа­лизованы достаточно быстро, за исключением первого. Именно разложе­ние числа  на простые множители и составляет наиболее трудоемкую часть вычислений. В теории чисел несмотря на многолетнюю её историю и на очень интенсивные поиски в течение последних 20 лет, эффективный алгоритм разложения натуральных чисел на множители так и не найден. Конечно, можно, перебирая все простые числа до , и. деля на них , найти требуемое разложение. Но, учитывая, что количество простых в этом промежутке, асимптотически равно , на­ходим, что при , записываемом 100 десятичными цифрами, найдётся не менее  простых чисел, на которые придётся делить  при разложе­нии его на множители. Очень грубые прикидки показывают, что компью­теру, выполняющему миллион делений в секунду, для разложения числа  таким способом на простые сомножители потребуется не менее, чем  лет. Известны и более эффективные способы разложения целых чисел на множители, чем простой перебор простых делителей, но и они работают очень медленно.

Авторы схемы RSA предложили выбирать число  в виде произведе­ния двух простых множителей  и , примерно одинаковых по величине. Так как

, (6)

то единственное условие на выбор показателя степени  в отображении (1) есть

. (7)

Итак, лицо, заинтересованное в организации шифрованной переписки с помощью схемы RSA, выбирает два достаточно больших простых числа  и . Перемножая их, оно находит число . Затем выбирается число , удовлетворяющее условиям (7), вычисляется с помощью (6) число  и с помощью (3) - число . Числа  и  публикуются, число  остается секретным. Теперь любой может отправлять зашифрованные с помощью (1) сообщения организатору этой системы, а организатор легко сможет расшифровывать их с помощью (5).

Для иллюстрации своего метода Ривест, Шамир и Адлеман зашифро­вали таким способом некоторую английскую фразу. Сначала она стан­дартным образом (а=01, b=02, .... z=26, пробел=00) была записана в виде целого числа , а затем зашифрована с помощью отображения (1) при

m=11438162575788886766932577997614661201021829672124236256256184293570 6935245733897830597123563958705058989075147599290026879543541

и . Эти два числа были опубликованы, причем дополнительно сообщалось, что . где  и  - простые числа, записываемые со­ответственно 64 и 65 десятичными знаками. Первому, кто расшифрует соответствующее сообщение

, была обещана награда в 100$.

Эта история завершилась спустя 17 лет в 1994 г., когда D. Atkins, M. Graff, А. К. Lenstra и Р. С. Leyland сообщили о расшифровке фразы. Числа  и  оказались равными

,

.

Этот замечательный результат (разложение на мно­жители 129-значного десятичного числа) был достигнут благодаря ис­пользованию алгоритма разложения чисел на множители, называемого методом квадратичного решета. Выполнение вычислений потребовало колоссальных ресурсов. В работе, возглавленной четырьмя авторами проекта, и продолжавшейся после предварительной теоретической под­готовки примерно 220 дней, на добровольных началах участвовало около 600 человек и примерно 1600 компьютеров, объединённых сетью Inter­net. Наконец, отметим, что премия в 100$ была передана в Free Software Foundation.

* 1. Сложность теоретико-числовых алгоритмов

Сложность алгоритмов теории чисел обычно принято измерять коли­чеством арифметических операций (сложений, вычитаний, умножений и делений с остатком), необходимых для выполнения всех действий, пред­писанных алгоритмом. Впрочем, это определение не учитывает величины чисел, участвующих в вычислениях. Ясно, что перемножить два стозначных числа значительно сложнее, чем два однозначных, хотя при этом и в том, и в другом случае выполняется лишь одна арифметическая опе­рация. Поэтому иногда учитывают ещё и величину чисел, сводя дело к так называемым битовым операциям, т. е. оценивая количество необхо­димых операций с цифрами 0 и 1, в двоичной записи чисел.

Говоря о сложности алгоритмов, мы будем иметь в ви­ду количество арифметических операций. При построении эффективных алгоритмов и обсуждении верхних оценок сложности обычно хватает ин­туитивных понятий той области математики, которой принадлежит алго­ритм. Формализация же этих понятий требуется лишь тогда, когда речь идёт об отсутствии алгоритма или доказательстве нижних опенок слож­ности.

Приведем теперь примеры достаточно быстрых алгоритмов с опен­ками их сложности. Здесь и в дальнейшем мы не будем придерживаться формального описания алгоритмов, стараясь в первую очередь объяснить смысл выполняемых действий.

Следующий алгоритм вычисляет  за  арифмети­ческих операций. При этом, конечно, предполагается, что натуральные числа  и  не превосходят по величине .

* + 1. Алгоритм вычисления 

1. Представим  в двоичной системе счисления , где , цифры в двоичном представлении, равны 0 или 1, .
2. Положим  и затем для  вычислим

.

3)  есть искомый вычет .

Справедливость этого алгоритма вытекает из сравнения

,

легко доказываемого индукцией по .

Так как каждое вычисление на шаге 2 требует не более трёх умноже­ний по модулю  и этот шаг выполняется  раз, то сложность алгоритма может быть оценена величиной .

Второй алгоритм - это классический алгоритм Евклида вычисления наибольшего общего делителя целых чисел. Мы предполагаем заданными два натуральных числа  и  и вычисляем их наибольший общий дели­тель .

* + 1. Алгоритм Евклида

1. Вычислим  - остаток от деления числа  на , , .
2. Если , то  есть искомое число.
3. Если , то заменим пару чисел  парой  и перейдем к  
   шагу 1.

Теорема 1. При вычислении наибольшего общего делителя  с помощью алгоритма Евклида будет выполнено не более  операций де­ления с остатком, где  есть количество цифр в десятичной записи меньшего из чисел  и .

Доказательство. Положим  и определим  - последовательность делителей, появляющихся в процессе выполнения ша­га 1 алгоритма Евклида. Тогда

.

Пусть также , , , , - последователь­ность Фибоначчи. Индукцией по  от  до  легко доказывается неравенство . А так как , то имеем неравенства  и .

Немного подправив алгоритм Евклида, можно достаточно быстро ре­шать сравнения  при условии, что . Эта задача равносильна поиску целых решений уравнения .

* + 1. Алгоритм решения уравнения 

1) Определим матрицу .

2) Вычислим  - остаток от деления числа  на , , .

3) Если , то второй столбец матрицы  даёт вектор   
решений уравнения.

4) Если , то заменим матрицу  матрицей .

5) Заменим пару чисел  парой  и перейдем к шагу 1.

Если обозначить  через матрицу , возникающую в процессе работы алгоритма перед шагом 3 после  делений с остатком (шаг 2), то в обозначениях из доказательства теоремы 1 в этот момент выполняется векторное равенство . Поскольку числа  и  взаимно просты, имеем , и это доказы­вает, что алгоритм действительно даёт решение уравнения . Буквой  мы обозначили количество делений с остатком, которое в точ­ности такое же, как и в алгоритме Евклида.

Три приведённых выше алгоритма относятся к разряду так называе­мых полиномиальных алгоритмов. Это название носят алгоритмы, слож­ность которых оценивается сверху степенным образом в зависимости от длины записи входящих чисел. Если наибольшее из чисел, подаваемых на вход алгоритма, не превосходит , то сложность алгоритмов этого типа оценивается величиной , где  - некото­рая абсолютная постоянная. Во всех приведённых выше примерах .

Полиномиальные алгоритмы в теории чисел - большая редкость. Да и опенки сложности алгоритмов чаше всего опираются на какие-либо не доказанные, но правдоподобные гипотезы, обычно относящиеся к анали­тической теории чисел.

Для некоторых задач эффективные алгоритмы вообще не известны. Иногда в таких случаях все же можно предложить последовательность действий, которая, «если повезет», быстро приводит к требуемому ре­зультату. Существует класс так называемых вероятностных алгоритмов, которые дают правильный результат, но имеют вероятностную опен­ку времени работы. Обычно работа этих алгоритмов зависит от одного или нескольких параметров. В худшем случае они работают достаточно долго. Но удачный выбор параметра определяет быстрое завершение ра­боты. Такие алгоритмы, если множество «хороших» значений параметров велико, на практике работают достаточно эффективно, хотя и не имеют хороших опенок сложности.

Мы будем иногда использовать слова детерминированный алгоритм, чтобы отличать алгоритмы в обычном смысле от вероятностных алго­ритмов.

Как пример, рассмотрим вероятностный алгоритм, позволяющий эф­фективно находить решения полиномиальных сравнений по простому мо­дулю. Пусть  — простое число, которое предполагается большим, и  - многочлен, степень которого предполагается ограничен­ной. Задача состоит в отыскании решений сравнения

. (8)

Например, речь может идти о решении квадратичных сравнений, если степень многочлена  равна 2. Другими словами, мы должны отыскать в поле  все элементы, удовлетворяющие уравнению .

Согласно малой теореме Ферма, все элементы поля  являются од­нократными корнями многочлена . Поэтому, вычислив наибольший общий делитель , мы найдем многочлен , множест­во корней которого в поле  совпадает с множеством корней многочлена , причем все эти корни однократны. Если окажется, что многочлен  имеет нулевую степень, т. е. лежит в поле , это будет означать, что сравнение (8) не имеет решений.

Для вычисления многочлена  удобно сначала вычислить многочлен , пользуясь алгоритмом, подобным описанному выше алгоритму возведения в степень (напомним, что число  предполагается большим). А затем с помощью аналога алгоритма Евклида вычислить . Всё это выполняется за полиномиальное количество арифметических операций.

Таким образом, обсуждая далее задачу нахождения решений сравне­ния (8), мы можем предполагать, что в кольце многочленов  спра­ведливо равенство:



* + 1. Алгоритм нахождения делителей многочлена  в кольце 

1) Выберем каким-либо способом элемент .

2) Вычислим наибольший общий делитель .

3) Если многочлен  окажется собственным делителем , то многочлен  распадётся на два множителя и с каждым из них незави­симо нужно будет проделать все операции, предписываемые настоящим алгоритмом для многочлена .

4) Если окажется, что  или , следует перейти к шагу 1 и. выбрав новое значение , продолжить выполнение алгоритма.

Количество операций на шаге 2 оценивается величиной , ес­ли вычисления проводить так, как это указывалось выше при нахожде­нии . Выясним теперь, сколь долго придётся выбирать числа , пока на шаге 2 не будет найден собственный делитель .

Количество решений уравнения  в поле  не превосходит . Это означает, что подмножество  элементов , удовлетворяющих условиям

,

состоит не менее, чем из  элементов. Учитывая теперь, что каждый ненулевой элемент  удовлетворяет одному из равенств , либо , заключаем, что для  одно из чисел  будет корнем многочлена , а другое - нет. Для таких элементов  многочлен , определённый на шаге 2 алгоритма, будет собственным делителем многочлена .

Итак, существует не менее  «удачных» выборов элемента , при которых на шаге 2 алгоритма многочлен  распадётся на два соб­ственных множителя. Следовательно, при «случайном» выборе элемента , вероятность того, что многочлен не разложится на множители после  повторений шагов алгоритма 1-4. не превосходит . Вероят­ность с ростом  убывает очень быстро. И действительно, на практике этот алгоритм работает достаточно эффективно.

Заметим, что при опенке вероятности мы использовали только два корня многочлена . При  эта вероятность, конечно, еще меньше. Более тонкий анализ с использованием опенок А. Вейля для сумм харак­теров показывает, что вероятность для многочлена  не распасться на множители при однократном проходе шагов алгоритма 1-4. не пре­восходит . Здесь постоянная в  зависит от .

Если в сравнении (8) заменить простой модуль  составным моду­лем , то задача нахождения решений соответствующего сравнения ста­новится намного более сложной. Известные алгоритмы её решения осно­ваны на сведении сравнения к совокупности сравнений (8) по простым модулям — делителям , и. следовательно, они требуют разложения чи­сла то на простые сомножители, что, как уже указывалось, является до­статочно трудоемкой задачей.

* 1. Качественная теория алгоритма RSA

Существует довольно эффективный способ убедиться, что заданное число является составным, не разлагая это число на множители. Согласно малой теореме Ферма, если число  простое, то для любого целого , не делящегося на , выполняется сравнение

. (9)

Если же при каком-то  это сравнение нарушается, можно утверждать, что  - составное. Проверка (9) не требует больших вычислений, это следует из алгоритма 1. Вопрос только в том, как найти для составного  целое число , не удовлетворяющее (9). Можно, например, пытаться найти необходимое число , испытывая все целые числа подряд, начиная с 2. Или попробовать выбирать эти числа случайным образом на отрезке .

К сожалению, такой подход не всегда даёт то, что хотелось бы. Име­ются составные числа , обладающие свойством (9) для любого целого  с условием . Такие числа называются числами Кармайкла. Рассмотрим, например, число . Так как 560 делится на каждое из чисел 2, 10, 16, то с помощью малой теоремы Ферма легко проверить, что 561 есть число Кармайкла. Можно доказать, что любое из чисел Кармайкла имеет вид , где все простые  различны, причем  делится на каждую разность . Лишь недавно, была решена проблема о бесконечности множества таких чисел.

В 1976 г. Миллер предложил заменить проверку (9) проверкой несколь­ко иного условия. Если  - простое число, , где  нечётно, то согласно ма­лой теореме Ферма для каждого  с условием  хотя бы одна из скобок в произведении



делится на . Обращение этого свойства можно использовать, чтобы от­личать составные числа от простых.

Пусть  - нечётное составное число, , где  нечётно. Назовем целое число , , «хорошим» для , если нарушается одно из двух условий:

1)  не делится на ;

2)  или существует целое , , такое, что

.

Из сказанного ранее следует, что для простого числа  не существует хороших чисел . Если же  составное число, то, как доказал Рабин, их существует не менее .

Теперь можно построить вероятностный алгоритм, отличающий со­ставные числа от простых.

* + 1. Алгоритм доказывающий не простоту числа

1. Выберем случайным образом число , , и проверим для  
   этого числа указанные выше свойства 1) и 2) п.2.
2. Если хотя бы одно из них нарушается, то число  составное.
3. Если выполнены оба условия 1) и 2) п.2, возвращаемся к шагу 1.

Из сказанного выше следует, что составное число не будет определено как составное после однократного выполнения шагов 1-3 с вероятностью не большей . А вероятность не определить его после  повторений не превосходит . т. е. убывает очень быстро.

Миллер предложил детерминированный алгоритм определения состав­ных чисел, имеющий сложность , однако справедливость его ре­зультата зависит от недоказанной в настоящее время так называемой расширенной гипотезы Римана. Согласно этому алгоритму достаточно проверить условия 1) и 2) п.2 для всех целых чисел , . Если при каком-нибудь  из указанного промежутка нарушается одно из условий а) или б), число  составное. В противном случае оно будет простым или степенью простого числа. Последняя возможность, конечно, легко проверяется.

Напомним некоторые понятия, необходимые для формулиров­ки расширенной гипотезы Римана. Они понадобятся нам и в дальнейшем. Пусть  - целое число. Функция  называется характе­ром Дирихле по модулю , или просто характером, если эта функция периодична с периодом , отлична от нуля только на числах, взаимно простых с , и мультипликативна, т. е. для любых целых  выполня­ется равенство . Для каждого  существует ровно  характеров Дирихле. Они образуют группу по умножению. Единичным элементом этой группы является так называемый главный характер , равный 1 на всех числах, взаимно простых с , и 0 на остальных це­лых числах. Порядком характера называется его порядок как элемента мультипликативной группы характеров.

С каждым характером может быть связана так называемая  - функция Дирихле - функция комплексного переменного , определённая рядом. Сумма этого ряда аналитична в области  и может быть аналитически продолжена на всю комплексную плос­кость. Следующее соотношение  связывает L - функцию, отвечающую главному характеру, с дзета-функцией Римана . Расширенная гипотеза Римана утверждает, что комплексные нули всех L -функций Дирихле, расположенные в полосе , лежат на прямой . В настоящее время не доказана даже простей­шая форма этой гипотезы - классическая гипотеза Римана, утвержда­ющая такой же факт о нулях дзета-функции.

В 1952 г. Анкени с помощью расширенной гипотезы Римана доказал, что для каждого простого числа  существует квадратичный невычет , удовлетворяющий неравенствам . Константа 70 была со­считана позднее. Именно это утверждение и лежит в основе алгоритма Миллера. В 1957 г. Берджесс доказал существование такого невычета без использования расширенной гипотезы Римана, но с худшей оценкой , справедливой при любом положительном  и , большем некоторой границы, зависящей от.

Алгоритм Миллера принципиально отличается от алгоритма 2.1., так как полученное с его помощью утверждение о том, что число  - со­ставное, опирается на недоказанную расширенную гипотезу Римана и по­тому может быть неверным. В то время как вероятностный алгоритм 2.1. даёт совершенно правильный ответ для составных чисел. Несмотря на отсутствие оценок сложности, на практике он работает вполне удовле­творительно.

* + 1. Нахождение больших простых чисел

Конечно же, большие простые числа можно строить сравнительно быстро. При этом можно обеспечить их случайное распределение в заданном диапазоне величин. В противном случае теряла бы всякий практический смысл система шифрования RSA. Наиболее эффективным средством построения простых чисел является несколько модифицированная малая теорема Ферма.

Теорема 2*.* Пусть  - нечётные натуральные числа, , причем для каждого простого делителя  числа  существует целое число  такое, что

. (10)

Тогда каждый простой делитель  числа  удовлетворяет сравнению

.

Доказательство. Пусть  - простой делитель числа , a  - не­который делитель . Из условий (10) следует, что в поле вычетов  спра­ведливы соотношения

. (11)

Обозначим буквой  порядок элемента  в мультипликативной группе поля . Первые два из соотношений (11) означают, что  входит в раз­ложение на простые множители числа  в степени такой же, как и в раз­ложение , а последнее - что  делится на . Таким образом, каждый простой делитель числа  входит в разложение  в степени не меньшей, чем в , так что  делится на . Кроме того,  четно. Теорема 2 доказана.

Следствие. Если выполнены условия теоремы 2 и , то  - простое число.

Действительно, пусть  равняется произведению не менее двух про­стых чисел. Каждое из них, согласно утверждению теоремы 2, не меньше, чем . Но тогда . Противоречие и доказывает следствие.

Покажем теперь, как с помощью последнего утверждения, имея боль­шое простое число , можно построить существенно большее простое число . Выберем для этого случайным образом чётное число  на про­межутке  и положим . Затем проверим число  на отсутствие малых простых делителей, разделив его на малые простые числа; испытаем  некоторое количество раз с помощью алгоритма 5. Если при этом выяснится, что  - составное число, следует выбрать новое значение  и опять повторить вычисления. Так следует делать до тех пор, пока не будет найдено число , выдержавшее испытания алго­ритмом 5 достаточно много раз. В этом случае появляется надежда на то, что  - простое число, и следует попытаться доказать простоту с помощью тестов теоремы 2.

Для этого можно случайным образом выбирать число , и проверять для него выполнимость соотношений

. (12)

Если при выбранном  эти соотношения выполняются, то, согласно след­ствию из теоремы 2, можно утверждать, что число  простое. Если же эти условия нарушаются, нужно выбрать другое значение  и повторять эти операции до тех пор, пока такое число не будет обнаружено.

Предположим, что построенное число  действительно является про­стым. Зададимся вопросом, сколь долго придётся перебирать числа , по­ка не будет найдено такое, для которого будут выполнены условия (12). Заметим, что для простого числа  первое условие (12), согласно малой теореме Ферма, будет выполняться всегда. Те же числа , для которых на­рушается второе условие (12), удовлетворяют сравнению . Как известно, уравнение  в поле вычетов  имеет не более  решений. Одно из них . Поэтому на промежутке  имеется не более  чисел, для которых не выполняются условия (12). Это означа­ет, что, выбирая случайным образом числа  на промежутке , при простом  можно с вероятностью большей, чем , найти чи­сло , для которого будут выполнены условия теоремы 2, и тем доказать, что  действительно является простым числом.

Заметим, что построенное таким способом простое число  будет удовлетворять неравенству , т. е. будет записываться вдвое боль­шим количеством цифр, чем исходное простое число . Заменив теперь число  на найденное простое число  и повторив с этим новым  все указанные выше действия, можно построить еще большее простое число. Начав с какого-нибудь простого числа, скажем, записанного 10 десятич­ными цифрами (простоту его можно проверить, например, делением на маленькие табличные простые числа), и повторив указанную процедуру достаточное число раз. можно построить простые числа нужной величи­ны.

Обсудим теперь некоторые теоретические вопросы, возникающие в связи с нахождением числа , удовлетворяющего неравенствам , и такого, что  - простое число. Прежде всего, со­гласно теореме Дирихле, доказанной еще в 1839 г., прогрессия ,  содержит бесконечное количество простых чисел. Нас интересуют простые числа, лежащие недалеко от начала прогрессии. Опенка наименьшего простого числа в арифметической прогрессии была полу­чена в 1944 г. Ю. В. Линником. Соответствующая теорема утверждает, что наименьшее простое число в арифметической прогрессии  не превосходит , где  - некоторая достаточно большая абсолютная постоянная.

Таким образом, в настоящее время никаких теоретических гарантий для существования простого числа  не сущес­твует. Тем не менее опыт вычислений на ЭВМ показывает, что простые числа в арифметической прогрессии встречаются достаточно близко к её началу. Упомянем в этой связи гипотезу о существовании бесконечного количества простых чисел  с условием, что число  также простое, т. е. простым является уже первый член прогрессии.

Очень важен в связи с описываемым методом построения простых чисел также вопрос о расстоянии между соседними простыми числами в арифметической прогрессии. Ведь убедившись, что при некотором  число  составное, можно следующее значение  взять равным  и действовать так далее, пока не будет найдено простое число . И если расстояние между соседними простыми числами в прогрессии ве­лико, нет надежды быстро построить нужное число . Перебор чисел  до того момента, как мы наткнемся на простое число  окажется слишком долгим. В более простом вопросе о расстоянии между соседни­ми простыми числами  и  в натуральном ряде доказано лишь, что , что, конечно, не очень хорошо для наших целей. Вместе с тем существует так называемая гипотеза Крамера (1936 г.), что , дающая вполне приемлемую опенку. Примерно такой же результат следует и из расширенной гипотезы Римана. Вычисления на ЭВМ показывают, что простые числа в арифметических прогрессиях расположены достаточно плотно.

В качестве итога обсуждения в этом пункте подчеркнём следующее: если принять на веру, что наименьшее простое число, а также расстояние между соседними простыми числами в прогрессии  при  оцениваются величиной , то описанная схема построения больших простых чисел имеет полиномиальную опенку сложности. Кро­ме того, несмотря на отсутствие теоретических опенок времени работы алгоритмов, отыскивающих простые числа в арифметических прогрес­сиях со сравнительно большой разностью, на практике эти алгоритмы работают вполне удовлетворительно. На обычном персональном компью­тере без особых затрат времени строятся таким способом простые числа порядка .

Конечно, способ конструирования простых чисел для использования в схеме RSA должен быть массовым, а сами простые числа должны быть в каком-то смысле хорошо распределёнными. Это вносит ряд дополнитель­ных осложнений в работу алгоритмов.

Наконец, отметим, что существуют методы построения больших про­стых чисел, использующие не только простые делители , но и делите­ли чисел . В основе их лежит использование после­довательностей целых чисел, удовлетворяющих линейным рекуррентным уравнениям различных порядков. Отметим, что последовательность , члены которой присутствуют в формулировке малой теоремы Ферма, со­ставляет решение рекуррентного уравнения первого порядка .

* + 1. Проверка большого числа на простоту

Есть некоторое отличие в постановках задач предыдущего и настоя­щего пунктов. Когда мы строим простое число , мы обладаем некото­рой дополнительной информацией о нем, возникающей в процессе постро­ения. Например, такой информацией является знание простых делителей числа . Эта информация иногда облегчает доказательство просто­ты .

В этом пункте мы предполагаем лишь, что нам задано некоторое чи­сло , например, выбранное случайным образом на каком-то промежут­ке, и требуется установить его простоту, или доказать, что оно является составным. Эту задачу за полиномиальное количество операций решает указанный в п. 3 алгоритм Миллера. Однако, справедливость полученного с его помощью утверждения зависит от недоказанной расширенной гипо­тезы Римана. Если число  выдержало испытания алгоритмом 5 для 100 различных значений параметра , то, по-видимому, можно утверждать, что оно является простым с вероятностью большей, чем . Эта вероятность очень близка к единице, однако всё же оставляет некоторую тень сомнения на простоте числа . В дальнейшем в этом пункте мы будем считать, что заданное число  является простым, а нам требуется лишь доказать это.

В настоящее время известны детерминированные алгоритмы различ­ной сложности для доказательства простоты чисел. Мы остановимся по­дробнее на одном из них, предложенном в 1983 г. в совместной работе Адлемана. Померанца и Рамели. Для доказательства простоты или непростоты числа  этот алгоритм требует  арифметиче­ских операций. Здесь  - некоторая положительная абсолютная посто­янная. Функция  хоть и медленно, но всё же возрастает с ростом , поэтому алгоритм не является полиномиальным. Но всё же его прак­тические реализации позволяют достаточно быстро тестировать числа на простоту. Существенные усовершенствования и упрощения в перво­начальный вариант алгоритма были внесены в работах X. Ленстры и А. Коена. Мы будем называть описываемый ниже алгоритм алго­ритмом Адлемана - Ленстры.

В основе алгоритма лежит использование сравнений типа малой те­оремы Ферма, но в кольцах целых чисел круговых полей, т. е. полей. порождённых над полем  числами  - корнями из 1. Пусть  - простое нечётное число и  — первообразный корень по модулю , т. е. образующий элемент мультипликативной группы поля , которая пиклична. Для каждого целого числа , не делящегося на , можно опре­делить его индекс, , называемый также *дискретным логарифмом*, с помощью сравнения . Рассмотрим далее два простых числа ,  с условием, что  делится на , но не делится на .

Следующая функция, определённая на множестве целых чисел.



является характером по модулю  и порядок этого характера равен .

Сумма:



называется суммой Гаусса. Формулируемая ниже теорема 3 представляет собой аналог малой теоремы Ферма, используемый в алгоритме Адлемана - Ленстры.

Теорема3. Пусть  - нечетное простое число, . Тогда в кольце  выполняется сравнение

.

Если при каких-либо числах  сравнение из теоремы 3 нарушается. можно утверждать, что  составное число. В противном случае, если сравнение выполняется, оно даёт некоторую информацию о возможных простых делителях числа . Собрав такую информацию для различных , в конце концов удаётся установить, что  имеет лишь один простой делитель и является простым.

В случае  легко проверить, что сравнение из теоремы 3 равно­сильно хорошо известному в элементарной теории чисел сравнению

, (13)

где  - так называемый символ Якоби. Хорошо известно также, что последнее сравнение выполняется не только для простых , но и для любых целых , взаимно простых с . Заметим также, что для вычисления символа Якоби существует быстрый алгоритм, основанный на законе вза­имности Гаусса и. в некотором смысле, подобный алгоритму Евклида вы­числения наибольшего общего делителя. Следующий пример показывает. каким образом выполнимость нескольких сравнений типа (13) даёт неко­торую информацию о возможных простых делителях числа .

Пример (X. Ленстра). Пусть  — натуральное число, . для которого выполнены сравнения

, (14)

а кроме того с некоторым целым числом  имеем

. (15)

Как уже указывалось, при простом  сравнения (14) выполняются для любого , взаимно простого с , а сравнение (15) означает, что  есть первообразный корень по модулю . Количество первообразных корней равно , т. е. достаточно велико. Таким образом, число  с усло­вием (15) при простом  может быть найдено достаточно быстро с помощью случайного выбора и последующей проверки (15).

Докажем, что из выполнимости (14-15) следует, что каждый делитель  числа  удовлетворяет одному из сравнений

 или . (16)

Не уменьшая общности, можно считать, что  - простое число. Введем теперь обозначения , где  и  - нечётные числа. Из (15) и сравнения  следует, что . Далее, согласно (14). выполняются следующие сравнения

,

означающие (в силу того, что символ Якоби может равняться лишь -1 или +1), что .

При  это равенство означает, что  при , и, следовательно, . Если же , то имеем  и . Этим (16) доказано.

Информация такого рода получается и в случае произвольных про­стых чисел  и  с указанными выше свойствами.

Опишем схему алгоритма Адлемана - Ленстры для про­верки простоты :

1. выбираются различные простые числа  и различные про­стые нечётные  такие, что
   1. для каждого  все простые делители числа  содержатся  
      среди  и  не делятся на квадрат простого числа;

1.2) .

1. для каждой пары выбранных чисел ,  проводятся тесты, подобные сравнению из теоремы 3. Если  не удовлетворяет какому-либо из  
   этих тестов - оно составное. В противном случае
2. определяется не очень большое множество чисел, с которыми толь­ко и могут быть сравнимы простые делители . А именно, каждый простой делитель  числа  должен удовлетворять сравнению вида

, .

1. проверяется, содержит ли найденное множество делители . Ес­ли при этом делители не обнаружены, утверждается, что  - простое  
   число.

Если число  составное, оно обязательно имеет простой делитель , меньший , который сам содержится среди возможных остатков. Именно на этом свойстве основано применение пункта 4) алгоритма.

Сумма Якоби:



определяется для двух характеров  модулю . Если характеры имеют порядок , то соответствующая сумма Якоби принадлежит кольцу . Поскольку числа , участвующие в алгоритме, сравнительно невели­ки, то вычисления с суммами Якоби производятся в полях существенно меньшей степени, чем вычисления с суммами Гаусса. Это главная при­чина, по которой суммы Якоби предпочтительнее для вычислений. При  выполняется классическое соотношение



связывающее суммы Гаусса с суммами Якоби и позволяющее переписать сравнение теоремы 3 в терминах сумм Якоби. Так. при  и  соответствующее сравнение, справедливое для простых , отлич­ных от 2,3,7, принимает вид

,

где  и  - некоторый корень кубический из 1.

В 1984 г. было внесено существенное усовершенствование в алгоритм, позволившее освободиться от требования неделимости чисел  на квадраты простых чисел. В результате, например, выбрав число  и взяв  равным произведению простых чисел  с условием, что  делится на , получим , что позволяет доказывать простоту чисел , записываемых сотней десятичных знаков. При этом вычисления будут проводиться в полях, порождённых корнями из 1 степеней 16, 9, 5 и 7.

Персональный компьютер с процессором Pentium-150. пользуясь реализацией этого алгоритма на языке UBASIC, доказал простоту записываемого 65 десятичными знаками, большего из простых чисел в при­мере Ривеста, Шамира и Адлемана за 8 секунд. Сравнение этих 8 секунд и 17 лет, потребовавшихся для разложения на множители предложенного в примере числа, конечно, впечатляет.

Отметим, что опенка сложности этого алгоритма представляет со­бой трудную задачу аналитической теории чисел. Как уже указывалось, количество операций оценивается величиной . Однако соот­ветствующие числа  и , возникающие в процессе доказательства, не могут быть явно указаны в зависимости от . Доказано лишь существо­вание чисел  и , для которых достигается оценка. Впрочем, есть веро­ятностный вариант алгоритма, доказывающий простоту простого числа  с вероятностью большей  за  арифметических операций. А в предположении расширенной гипотезы Римана эта опенка сложности может быть получена при эффективно указанных  и.

Выводы по главе

1. Проектирование

Разрабатываемую информационную систему можно отнести к клиент-серверной архитектуре. На стороне клиента используется приложение, написанное под операционную систему GNU/Linux Debian Wheezy на языке Python.

* 1. Проектирование функций
* rsa.**encrypt**(message, pub\_key)

Шифрует данное сообщение используя PKCS#1 v1.5

**Parameters:**

1. **message** – сообщение для шифрования. Должен быть строкой не более k-11, где k – число байтов, для кодирования n компонент открытого ключа.
2. **pub\_key** –открытый ключ rsa.PublicKey для шифрования.

**Выдает OverflowError:**

Сообщение слишком велико, чтобы уместить блок кодирования.

* rsa.**decrypt**(*crypto*, *priv\_key*)

Расшифровывает данное сообщение с помощью PKCS # 1 v1.5

Расшифровка считается "не удалась", когда в результате открытый текст не начинается с байтов 00 02, или когда 00 байт между внутренним отступом и сообщением, не могут быть найдено.

**Parameters:**

1. **crypto** – крипто текст как результат выполнения rsa.PublicKey()
2. **priv\_key** – закрытый ключ rsa.PrivateKey для дешифрования.

**Выдает DecryptionError:**

Когда не поддается расшифровке. Никаких подробностей не приводится, почему не получилось расшифровать, т.к. это будет наводящей информацией о закрытом ключе.

* rsa.**sign**(message, priv\_key, hash)

Подписывает сообщение с секретным ключом.

Хэширует сообщение, затем подписывает хэш с данным ключом. Это известно как «Отдельная сигнатура», потому что само сообщение не изменяется.

**Parameters:**

1. **message** – подписываемое сообщение
2. **priv\_key** – закрытый ключ rsa.PrivateKey для подписывания сообщения.
3. **hash** – способ хеширования сообщения. Возможно использовать ‘MD5’, ‘SHA-1’, ‘SHA-256’, ‘SHA-384’ или ‘SHA-512’.

**Возвращает:**

Блок подписанного сообщения.

**Выдает OverflowError:**

Если закрытый ключ слишком мал, чтобы содержать выбранный хэш.

* rsa.**verify**(message, signature, pub\_key)

Проверяет подпись сообщения на совпадение.

Хэш метод определяется автоматически с момента подписания.

**Parameters:**

1. **message** – подписанное сообщение. Может быть 8-битной строкой или файл-подобный объект.
2. **signature**  – блок подписи, созданный с rsa.sign()
3. **pub\_key** – публичный ключ rsa.PublicKey подписавшего сообщение.

**Выдает VerificationError:**

Когда подпись не совпадает с сообщением.

* rsa.**newkeys**(keysize)

Создает открытый и закрытый ключи и возвращает их в виде (открытый, закрытый).

Открытый ключ нужен как «ключ шифрования» и является объектом rsa.PublicKey. Закрытый ключ нужен как «ключ дешифрования» и является объектом rsa.PrivateKey.

**Parameters:**

1. **nbits** – количество бит необходимых для хранения n = p \* q
2. **accurate**  – когда истина, n будет иметь запрошенное количество битов.
3. **poolsize** – количество используемых процессов для генерации простых чисел. Если установлено число большее 1, параллельный алгоритм будет использоваться. Это требует Python 2.6 и новее.

**Возвращает:**

Запись (rsa.PublicKey, rsa.PrivateKey)

* 1. Проектирование классов
* class rsa.**PublicKey**(n, e)

Представляет открытый ключ RSA

Этот ключ используется как «ключ шифрования». Он содержит ‘N’ и ‘E’ значения.

Поддерживает атрибуты, а также словарь доступа. Атрибуты доступа все же быстрее.

**load\_pkcs1**(keyfile, format='PEM')

Загружает ключ PKCS#1 в DER или PEM форматах.

**Parameters:**

1. **keyfile** – содержимое DER или PEM закодированного файла, содержащего открытый ключ.
2. **format**  – загружаемый формат файла DER или PEM.

**Возвращает:**

Объект публичного ключа PublicKey.

classmethod **load\_pkcs1\_openssl\_pem**(keyfile)

Загружает файл открытого ключа PKCS#1 в DER – кодировке из OpenSSL

**Parameters:**

Параметр **keyfile** – содержимое файла в DER кодировке, что содержит открытый ключ OpenSSL

**Возвращает:**

Объект публичного ключа PublicKey.

classmethod **load\_pkcs1\_openssl\_der**(n, e, d, p, q, exp1=None, exp2=None, coef=None)

Загружает файл открытого ключа PKCS#1 в PEM – кодировке из OpenSSL

Этот файл признается в том случае, если начинается с BEGIN PUBLIC KEY, а не с BEGIN RSA PUBLIC KEY.

Содержимое файла перед “—–BEGIN PUBLIC KEY—–” и после “—–END PUBLIC KEY—–” линий, игнорируются.

**Parameters:**

Параметр **keyfile** – содержимое файла в PEM кодировке, что содержит открытый ключ OpenSSL

**Возвращает:**

Объект публичного ключа PublicKey.

**save\_pkcs1**(format='PEM')

Сохраняет публичный ключ PKCS#1 в DER или PEM формате

**Parameters:**

**format** – формат сохранения (DER или PEM формат).

**Возвращает:**

Публичный ключ шифрования в DER или PEM формате.

* class rsa.PrivateKey(n, e, d, p, q, exp1=None, exp2=None, coef=None)

Представляет секретный ключ RSA.

Этот ключ используется как «ключ дешифрования». Он содержит ‘n’, ‘e’, ‘d’, ‘p’, ‘q’ и другие значения.

Поддерживает атрибуты, а также словарь доступа. Атрибут доступа все же быстрее.

**load\_pkcs1**(keyfile, format='PEM')

Загружает ключ PKCS#1 в DER или PEM формате

**Parameters:**

1. **keyfile** – содержимое в DER или PEM формате закодированного файла, содержащего открытый ключ.
2. **format -** формат файла для загрузки (DER или PEM формат).

**Возвращает:**

Объект публичного ключа.

**save\_pkcs1**(format='PEM')

Сохраняет публичный ключ PKCS#1 в DER или PEM формате

**Parameters:**

1. **format -** формат файла для сохранения (DER или PEM формат).

**Возвращает:**

Публичный ключ в DER или PEM формате.

* 1. Проектирование классов исключений
* *class* rsa.pkcs1.**CryptoError**(Exception)

Базовый класс для всех исключений в этом модуле.

* *class* rsa.pkcs1.**DecryptionError**(CryptoError)

Возникает, когда расшифровать не удается.

* *class* rsa.pkcs1.**VerificationError**(CryptoError)

Возникает, когда верифицировать не удается.

* 1. Модуль rsa.bigfile

Модуль rsa.bigfile содержит функции для шифрования и дешифрования файлов, размер которых превышает размер ключа RSA.

rsa.bigfile.**encrypt\_bigfile**(infile, outfile, pub\_key)

Шифрует файл, записывая его в выходной файл «outfile» в формате VARBLOCK

**Parameters:**

1. **infile –** файл-подобный объект, чтобы читать открытым ткстом.
2. **outfile** – файл-подобный объект, чтобы зашифровать в формате VARBLOCK.
3. **pub\_key** – публичный ключ шифрования rsa.PublicKey

rsa.bigfile.**decrypt\_bigfile**(infile, outfile, priv\_key)

Дешифрует зашифрованный VARBLOCK файл, записывая в входной файл «outfile».

**Parameters:**

1. **infile –** файл-подобный объект, чтобы прочитать шифрованный VARBLOCK файл.
2. **outfile** – файл-подобный объект, чтобы читать открытым ткстом.
3. **priv\_key** – закрытый ключ дешифрования rsa.PrivateKey
   1. Формат VARBLOCK файла

Формат файла VARBLOCK позволяет шифровать файлы, размер которых превышает размер ключа RSA. Формат должен быть следующим; || обозначает конкатенацию байт строк:

VARBLOCK := VERSION || BLOCK || BLOCK || ...

VERSION := 1

BLOCK := LENGTH || DATA

LENGTH := varint-кодировка длины в байтах

DATA := данные для хранения в блоке

Varint-формат был взят из Protobuf Google, и позволяет эффективно кодировать сколь угодно длинное целое.

* 1. Модуль rsa.core

В основе метода шифрования RSA лежат эти две функции. Они обе работают только на (сколько угодно длинных) целых числах. Эти функции являются ядром всей библиотеки.

rsa.core.**encrypt\_int**(message, ekey, n)

Шифрует сообщения с использованием ключа шифрования «Ekey» по модулю n.

rsa.core.**decrypt\_int**(cyphertext, dkey, n)

Дешифрует зашифрованный текст с помощью дешифрующего ключа «dkey» по модулю n.

Выводы по главе

1. Тестирование и документирование
   1. Тестирование

Проверка качества проекта и информационной системы. Поиск и исправление ошибок после распространения продукта обходится значительно дороже, чем до распространения, поэтому тестирование должно быть максимально полным.

Тестирование позволяет:

* объективно оценить состояние системы;
* выявить противоречия в требованиях;
* проверить устойчивость к риску;
* выявить дефекты в проектировании на ранних этапах;
* применять автоматизированные средства тестирования;

Контроль изменения предполагает:

* непрерывное наблюдение за ходом разработки системы;
* координацию изменений на этапах завершения очередной версии и устранение несогласованности путем возврата на предыдущие этапы разработки.
* непрерывное накопление возможных изменений и поэтапное их внедрение или отклонение.

В тестирование программ можно выделить четыре этапа:

* тестирование отдельных модулей;
* совместное тестирование модулей;
* тестирование спецификации программы;
* тестирование всего комплекса в целом (т.е. поиск несоответствия созданного программного продукта сформулированным ранее целям проектирования, отраженным в техническом задании).

На первых двух этапах используются, прежде всего, методы структурного тестирования, т.к. на последующих этапах тестирования эти методы использовать сложнее из-за больших размеров проверяемого программного обеспечения; последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, которые не обязательно связаны с логикой программы.

Известны два подхода к тестированию модулей: монолитное и пошаговое тестирование.

* 1. Руководство программиста
     1. Назначение и область применения

Библиотека алгоритма шифрования решает основные проблемы в области защищенной передачи данных:

* Генерация псевдо-случайных чисел;
* Подписывание данных сертификатом;
* Проверка сертификатов;
* Установка защищенного соединения;
* Шифрование или дешифрование неограниченно больших файлов;

В библиотеке алгоритма шифрования версии 1.1 предусмотрены возможности автоматической настройки с рекомендуемыми параметрами. Данная возможность позволяет использовать алгоритм шифрования с минимальными настройками, что в свою очередь повышает надежность и предостерегает от неопытного использования.

* + 1. Характеристики программы

Размер программы 109 Кб, размер всех файлов, включая код тестирования программы, а также файлов установки составляет 132 Кб.

В каталоге запуска программы должны быть файлы:

* key\_priv.rsa – файл с приватным ключом. Используется для личного использования. Этот файл нельзя передавать или открывать доступ к этому файлу в целях безопасности.
* key\_pub.rsa – файл с публичным ключом. Открытый ключ нужен для шифрования передаваемых данных, его можно передавать другим.

Для выполнения программы необходима следующая минимальная конфигурация оборудования:

* процессор с частотой не менее 700 МГц;
* жесткий диск с размером не менее 2Гб;
* оперативная память не менее 128Мб;
* клавиатура
* сетевой адаптер

Для функционирования библиотеки алгоритмов необходима предварительная установка следующих компонентов:

* Python не ниже версии 2.6
* VirtualEnv
* PIP
* GNU/Linux Debian Wheezy

В случае отсутствия необходимых пакетов на компьютере пользователя, в директории с инсталляционным файлом программы имеется все необходимое для установки. Инсталляционный пакет автоматически выполнит проверку, и установит все необходимое.

* + 1. Обращение к программе

Открытый ключ используется для шифрования сообщения так, чтобы он мог быть прочитан только владельцем закрытого ключа. Расшифровка сообщения может быть выполнена только с помощью закрытого ключа, поэтому он называется ключ дешифрования.

Закрытый ключ используется для подписи сообщения. С помощью открытого ключа и подписи, приемник может проверки того, что сообщение было подписано владельцем закрытого ключа, и что сообщение не было изменено после подписания.

* + - 1. Генерация ключа

Для генерации ключа можно использовать два способа:

* генерация с помощью функции rsa.newkeys(), пример на рисунке 4.1.
* альтернативная генерация с помощью rsa.PrivateKey.load\_pkcs1() и rsa.PublicKey.load\_pkcs1(), пример на рисунке 4.2.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\Screenshot-3215.jpg

Рисунок 4.1. Генерация ключа с помощью функции rsa.newkeys()

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\Screenshot-3214.jpg

Рисунок 4.2. Альтернативная генерация ключа.

* + - 1. Время генерации ключей

Генерирование ключей может занять длительное время, в зависимости от необходимого числа битов,. Число битов определяет криптостойкость ключа, а также размером сообщения, которое нужно зашифровать. Есть возможность, немного уменьшить размер ключа, указав параметр accurate= False, чтобы ускорить процесс генерации ключа.

Еще один способ ускорить процесс генерации ключа - параллельное использование нескольких процессов для ускорения генерации ключей. Стоит указывать число не больше, чем есть на самом деле в процессоре используемого оборудования; двухъядерная машина должна использовать PoolSize = 2; на четырехъядерном процессоре с двумя потоками, на каждом ядре, можно использовать PoolSize = 8.

В таблице 4.1 «Скорость генерации ключей» на разных процессорах с использованием 64-разрядного CPython 2.7. Поскольку генерация ключа является случайным процессом, результаты могут отличаться даже на аналогичном оборудовании. На всех тестах использован параметр accurate= True.

Таблица 4.1. Скорость генерации ключей.

|  |  |  |
| --- | --- | --- |
| **Keysize (бит)** | **Один поток** | **8 потоков** |
| 128 | 0.01 sec. | 0.01 sec. |
| 256 | 0.03 sec. | 0.02 sec. |
| 384 | 0.09 sec. | 0.04 sec. |
| 512 | 0.11 sec. | 0.07 sec. |
| 1024 | 0.79 sec. | 0.30 sec. |
| 2048 | 6.55 sec. | 1.60 sec. |
| 3072 | 23.4 sec. | 7.14 sec. |
| 4096 | 72.0 sec. | 24.4 sec. |

Если генерация ключей занимает слишком много времени, тогда можно в качестве альтернативы использовать OpenSSL а затем загрузить их в коде Python. OpenSSL генерирует 4096-битный ключ за 3,5 секунды на той же машине, используемой выше.

* + - 1. Требования к размеру ключа

Python-RSA версии 3.0 описывает PKCS # 1 в стиле случайного заполнения. Это означает, что 11 байтный (88 битный) сгенерированный ключ не пригоден для шифрования. Чем больше размер ключа, тем выше стойкость к атакам. Таблица 4.2.

Для создания подписи также требуется ключ определенного размера, в зависимости от используемого метода хеширования:

Таблица 4.2. Минимальные требования к длине генерируемого ключа.

|  |  |
| --- | --- |
| **Метод хеширования** | **Допустимый минимальный**  **размер ключа (бит)** |
| MD5 | 360 |
| SHA-1 | 368 |
| SHA-256 | 496 |
| SHA-384 | 624 |
| SHA-512 | 752 |

* + - 1. Шифрование и дешифрование

Чтобы зашифровать или расшифровать сообщение, используйтся rsa.encrypt() и rsa.decrypt() соответственно.

К примеру предположим, что Алиса хочет послать сообщение, так чтобы только Боб смог его прочитать.

1. Боб создает пару ключей, и дает открытый ключ Алисе. Это делается таким образом, чтобы Алиса точно знала, что ключ действительно принадлежит Бобу. Рисунок 4.3.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.4\Screenshot-3216.jpg

Рисунок 4.3. Создание новой пары ключей.

1. Алиса пишет сообщение. Рисунок 4.4.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.4\Screenshot-3217.jpg

Рисунок 4.4. Создание сообщения.

1. Алиса шифрует сообщение используя открытый ключ Боба и посылает зашифрованное сообщение. Рисунок 4.5.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.4\Screenshot-3218.jpg

Рисунок 4.5. Шифрование сообщения.

1. Боб получает сообщение и расшифровывает его с помощью своего закрытого ключа. Рисунок 4.6.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.4\Screenshot-3219.jpg

Рисунок 4.6. Дешифрование сообщения.

Поскольку Боб держал закрытый приватный ключ, Алиса может быть уверена, что он единственный, кто может прочитать её сообщение. Боб не знает наверняка, что это была Алиса, которая отправила сообщение, так как она не подписала его.

RSA может только шифровать сообщения, которые меньше, чем ключ. Пару байт теряются на случайные заполнения, а остальное уже доступны для самого сообщения. Например, 512-битный ключ может кодировать сообщение 53-байт (512 бит = 64 байт, 11 байт используются для случайного заполнения и другие вещи).

Альтернативное шифрование информации, скорее всего, может привести к возникновению исключения rsa.pkcs1.DecryptionError. Желательно использовать rsa.sign (). Рисунок 4.7.



Рисунок 4.7. Исключительная ситуация.

Никогда не стоит отображать трассировку стека исключения rsa.pkcs1.DecryptionError. Это покажет, где в коде произошла ошибка, и, таким образом, может произойти утечка информации о ключе. Это всего лишь крошечная часть информации, но каждый бит повышает вероятность взлома ключа.

* + - 1. Низкоуровневые операции

Основной алгоритм RSA поддерживает большие целые числа. Эти операции считаются низкоуровневыми и поддерживаются функциями rsa.core.encrypt\_int() и rsa.core.decrypt\_int().

* + - 1. Подписывание и проверка

Можно создать отдельную подпись для сообщения, используя функцию rsa.sign(). Рисунок 4.8.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.5\Screenshot-3221.jpg

Рисунок 4.8. Подписывание сообщения.

Этот хэш сообщения, использует алгоритм SHA-1. Затем результат вычисления хэш-функции был подписан закрытым ключом. Также возможно использовать другие хэш методы.

Для проверки подписи, существует функция rsa.verify(). Эта функция возвращает истину, если проверка проходит успешно. Рисунок 4.9.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.6\Screenshot-3222.jpg

Рисунок 4.9. Проверка подписанного сообщения и открытого ключа.

Изменение сообщения или подписи приведет к ошибке верификации и вызовет исключение rsa.pkcs1.VerificationError. Рисунок 4.10.

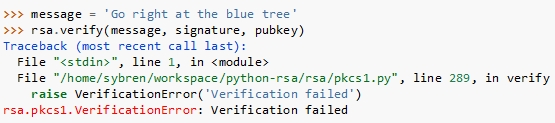


Рисунок 4.10. Ошибка верификации.

Вместо сообщения, можно также вызывать функции rsa.sign() и rsa.verify() с файл-подобным объектом. Если объект сообщения имеет метод read(int), тогда предполагается, что этот объект был построен из файла. В этом случае файл хешируется в 1024-байтные блоки. Рисунок 4.11 и рисунок 4.12.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.6\Screenshot-3224.jpg

Рисунок 4.11.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.6\Screenshot-3225.jpg

Рисунок 4.12.

* + - 1. Работа с большими файлами

RSA может шифровать только сообщения, длина которых меньше, чем длина ключа. Пару байт теряются на случайные заполнения, а остальное используется для самого сообщения. Например, 512-битный ключ может кодировать сообщение 53-байт (512 бит = 64 байт, 11 байт используются для случайного заполнения).

* + - * 1. Применение

Самый распространенный способ использования RSA с большими файлами - это использование блочного шифра, таких как AES или DES3 для шифрования файлов со случайным ключом, предварительно передав этот ключ по защищенному каналу, установленного с помощью RSA. Пример:

1. Генерация случайного ключа. Рисунок 4.13.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.1\Screenshot-3226.jpg

Рисунок 4.13. Генерация случайного ключа.

1. Использование этого ключа для шифрования файла с помощью AES.
2. Шифрование ключа AES с помощью RSA. Рисунок 4.14.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.1\Screenshot-3227.jpg

Рисунок 4.14. Шифрование ключа AES.

1. Отправка шифрованного файла вместе с encrypted\_aes\_key.
2. Получатель обращает этот процесс, чтобы получить зашифрованный файл.

Модуль Python-RSA не содержит функцию шифрования AES.

* + - * 1. Использование Python-RSA: VARBLOCK формат

Предыдущие версии Python-RSA содержали функции для шифрования больших файлов с помощью RSA. Текущая версия была улучшена, однако нет функции Python AES.

Шифрование работает следующим образом: входной файл разбивается на блоки, которые достаточно большие, но позволяют шифровать с ключом RSA. Затем каждый блок шифруется с помощью RSA. После все зашифрованные блоки собираются в выходной файл. Этот формат называется форматом VARBLOCK.

Расшифровка работает в обратном направлении. Зашифрованный файл разделяется на зашифрованные блоки. Потом каждый блок расшифровывается и обратно собираются в большой файл.

Зашифрованный файл получится больше исходного по размеру, так как каждый зашифрованный блок имеет 8 байт случайно заполненного места и еще 3 байта для дополнительной информации.

Так как функции шифрования/дешифрования предназначены для очень больших файлов, они используют другой подход. Там, где регулярные функции хранят сообщение в памяти на протяжении всего времени, эти функции работают над одним блоком в определенный момент времени. В результате, необходимо вызывать их с файл-подобными объектами в виде параметра.

Перед использованием, конечно, нужно сгенерировать пару ключей keypair. Рисунок 4.15.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.2\Screenshot-3228.jpg

Рисунок 4.15. Генерация новой пары ключей.

Шифрование работает с дескрипторами файлов с помощью функции rsa.bigfile.encrypt\_bigfile(). Рисунок 4.16.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.2\Screenshot-3229.jpg

Рисунок 4.16. Шифрование большого файла.

Дешифрование с помощью функции rsa.bigfile.decrypt\_bigfile(). Рисунок 4.17.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.2\Screenshot-3230.jpg

Рисунок 4.17. Дешифрование большого файла.

Функции rsa.sign() и rsa.verify() работают на произвольно длинных файлах, поэтому они не имеют «bigfile» реализации.

* + - * 1. Формат VARBLOCK

Этот раздел описывает двоичный формат связи для протокола буферных сообщений. Нет большой необходимости знать как использовать буфер протокол в своих приложениях, но очень полезно знать, как различные форматы буфера протокола влияют на размер закодированных сообщений.

Простое сообщение

При необходимости зашифровать простое сообщение (рисунок 4.18), в созданном приложении, создается сообщение Test1 и устанавливается до 150. Затем серийно посылаются сообщение в выходной поток. В результате получится три байта. Рисунок 4.19.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.1\Screenshot-3231.jpg

Рисунок 4.18.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.1\Screenshot-3232.jpg

Рисунок 4.19. Результат выполнения.

Варианты формата base 128

При кодировании буфера протокола, в первую очередь необходимо понять, varints. Varints - это метод сериализации целых чисел, используя один или несколько байт. Меньшее число принимает меньшее количество байт.

Каждый байт в varint, за исключением последнего байта, имеет самый старший бит (MSB) - это означает, что есть еще байт. Младшие 7 бит каждого байта используются для хранения в дополнение до двух представленных чисел в группах по 7 бит, наименее значительна первая группа.

Так, например, номером 1 - это один байт, поэтому старший бит не установлен. Рисунок 4.20.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.2\Screenshot-3233.jpg

Рисунок 4.20.

Для 300 необходимо немного больше. Рисунок 4.21.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.2\Screenshot-3234.jpg

Рисунок 4.21.

Чтобы определить, что это 300, нужно сначала опустить MSB от каждого байта, так как только это поможет определить достигли ли конца числа. Рисунок 4.22.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.2\Screenshot-3235.jpg

Рисунок 4.22.

Далее нужно инвертировать две группы по 7 бит, потому что varints хранит номера в первую очередь с наименьшим значением группы. После нужно объединить их, чтобы получить окончательное значение. Рисунок 4.23.

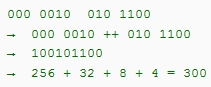


Рисунок 4.23.

Структура сообщений

Протокол буферных сообщений представляет собой серию пар ключ-значение. В бинарных сообщениях просто используется номер поля в качестве ключа - название и объявленный тип для каждого поля можно определить только в конце декодирования ссылаясь на определение типа сообщения (то есть файл .proto).

Когда сообщение кодируется, ключи и значения объединяются в поток байтов. Когда сообщение декодируется, анализатор должен пропустить поля, которые не может распознать. Таким образом, новые поля могут быть добавлены в сообщение, не нарушая целостность структуры для старых программ, которые не умеют с этим работать. С этой целью, «ключ» для каждой пары в сообщении связь-формат на самом деле два значения - значение поля сиз файла .proto, а также тип связи, что обеспечивает достаточно информации, чтобы найти длину следующего значения.

Возможные типы связей указаны в таблице 4.3.

Таблица 4.3.

|  |  |  |
| --- | --- | --- |
| **Тип** | **Значение** | **Используется для** |
| 0 | Varint | int32, int64, uint32, uint64, sint32, sint64, bool, enum |
| 1 | 64-bit | fixed64, sfixed64, double |
| 2 | Length-delimited | string, bytes, embedded messages, packed repeated fields |
| 3 | Start group | groups (устаревший) |
| 4 | End group | groups (устаревший) |
| 5 | 32-bit | fixed32, sfixed32, float |

Каждый ключ в потоковом сообщения является varint со значением (field\_number << 3) | wire\_type - иными словами, последние три бита числа хранят типа связи.

Первое число в потоке всегда является ключевым varint, и здесь эти 08 или MSB. Рисунок 4.24.

F:\Diplom\Защита протокола обмена данных. Олег\Diplom\Картинки\4.2.3.7.3.3\Screenshot-3240.jpg

Рисунок 4.24.

Последние три бита нужны, чтобы получить тип связи(0) и тогда сдвиг вправо на три даст номер поля(1). Таким образом, тег является 1 и следующее значение является varint. Используя varint-декодирования можно увидеть, что в ближайших двух байтах хранится значение 150. Рисунок 4.25.

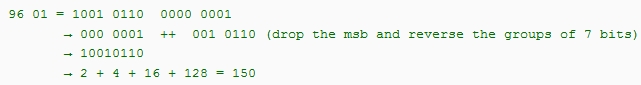


Рисунок 4.25.

* 1. Руководство пользователя
     1. Общие сведения о программе

*Обозначение и наименование программы*

Обозначение программы: «rsa.py»

Наименование программы: «Библиотека алгоритма шифрования»

*Язык программирования, на котором написана программа*

Программа написана на языке Python

*Назначение и возможности программы*

* Программа предназначена для защиты протокола обмена данных.
* Программа «Обозреватель контрактов» версии 1.0 выполняет следующие основные функции:
* Генерация псевдо-случайных чисел;
* Подписывание данных сертификатом;
* Проверка сертификатов;
* Установка защищенного соединения;
* Шифрование или дешифрование неограниченно больших файлов;
* Применять сторонние алгоритмы шифрования для большей защиты

*Ограничения области применения программы*

Программа «Библиотека алгоритма шифрования» версии 1.1 может быть применима только для защиты протокола обмена данными в качестве модуля для системы «Звезда» версии 1.2. Для использования программы в других системах не предусмотрены функции взаимодействия, что приводит к необходимости создания дополнительных функций, с учетом специфичных особенностей других систем.

*Сведения о функциональных ограничениях на применение*

Для функционирования программы «Библиотека алгоритма шифрования» версии 1.0 необходима предварительная установка следующих компонентов:

* Python не ниже версии 2.6
* VirtualEnv
* PIP
* GNU/Linux Debian Wheezy.

В случае отсутствия необходимых пакетов на компьютере пользователя, в директории с инсталляционным файлом программы имеется все необходимое для установки. Инсталляционный пакет автоматически выполнит проверку, и установит все необходимое.

* + 1. Структура системы
  1. Руководство администратора

Вывод по главе

1. Экономическое обоснование разработки
   1. Описание работы
   2. Описание конкурентов
   3. Описание рынка сбыта

5.4. Расчет стоимости программной продукции

* + 1. Расчет трудоемкости

Трудоемкость разработки программной продукции зависит от ряда факторов, основными из которых являются: степень новизны разрабатываемого программного комплекса, сложность алгоритма его функционирования, объем используемой информации, вид ее представления и способ обработки, а также уровень используемого алгоритмического языка программирования.

Функциональным назначением разработанной в дипломном проекте программной продукции является управление документопотоком.

По степени новизны программная продукция относится к группе B (разработка программной продукции, имеющей аналоги).

По степени сложности алгоритма функционирования программная продукция относится к группе 3 (программная продукция, реализующая алгоритмы стандартных методов решения задач).

Используемая информация представлена в виде базы данных. Учитывая вид представления, способ контроля и структуру выходных документов, исходную информацию можно отнести к группе 1.1 – исходная информация представлена в форме документов, имеющих различный формат и структуру, требуется учитывать взаимовлияние показателей в различных документах. Структура выходных документов требует вывод на печать документов многоуровневой структуры - группа 2.1.

Количество разновидностей форм входной информации равно одному виду (информация об абитуриентах), количество разновидностей форм выходной информации также равно одному (отчеты) [11].

Трудоемкость разработки программной продукции  может быть определена как сумма величин трудоемкости выполнения отдельных стадий разработки ПП из выражения (5.1):

, (5.1)

где - трудоемкость разработки технического задания на создание ПП;  - трудоемкость разработки эскизного проекта ПП;  - трудоемкость разработки технического проекта ПП;  - трудоемкость разработки рабочего проекта ПП;  - трудоемкость внедрения разрабатываемого ПП.

Трудоемкость разработки технического задания рассчитывается по формуле(5.2):

, (5.2)

где  - затраты времени разработчика постановки задач на разработку ТЗ, чел.дн.;  - затраты времени разработчика ПО на разработку ТЗ, чел.дн.

Значения величин  и  рассчитывают по формулам (5.3) и (5.4)

, (5.3)

, (5.4)

где  - норма времени на разработку ТЗ на программный продукт в зависимости от функционального назначения и степени новизны разрабатываемого ПП, чел.дн.;  - коэффициент, учитывающий удельный вес трудоемкости работ, выполняе­мых разработчиком постановки задач на стадии ТЗ (в случае сов­местной с разработчиком ПО разработки =0,65);  - коэффи­циент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком ПО на стадии ТЗ (в случае совместной с разработчиком постановки задач =0,35).

Для разработанного ПП норма времени =24 [2] (в данном случае 12), следовательно, после подстановки значений в (5.2), (5.3) и (5.4), получается:







Трудоемкость разработки эскизного проекта ПП  рассчитывают по формуле (5.5):

, (5.5)

где  - затраты времени разработчика постановки задач на разработку ЭП, чел.дн.;  - затраты времени разработчика ПО на разработку ЭП, чел.дн.

Значения величин и  рассчитывают по формулам (5.6) и (5.7):

, (5.6)

, (5.7)

где  - норма времени на разработку ЭП программного продукта в зависимости от его функционального назначения и степени новизны, чел.дн.;  – коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком постановки задач на стадии ЭП (в случае сов­местной с разработчиком ПО разработки  = 0,7);  - коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком ПО на стадии ЭП (в случае совместной с разработчиком постановки задач =0,3).

Для разработанного ПП норма времени  = 70 [2] (в данном случае 20), следовательно, после подстановки значений в (5.5), (5.6) и (5.7), получается:







Трудоемкость разработки технического проекта  зависит от функционального назначения ПП, количества разновидностей форм входной и выходной информации и определяется как сумма времени, затраченного разработчиком постановки задач и разработчиком программного обеспечения (5.8):

*,* (5.8)

где ,  – норма времени, затрачиваемого на разработку ТП разработчиком постановки задач и разработчиком ПО соответственно, чел.дн.;  – коэффициент учета вида используемой информации;  – коэффициент учета режима обработки информации.

Значение коэффициента  определяют из выражения (5.9):

, (5.9)

где , ,  - значения коэффициентов учета вида используемой информации для переменной, нормативно-справочной инфор­мации и баз данных соответственно; , ,  – количество наборов данных переменной, нормативно-справочной информации и баз данных соответственно.

Для разработанного ПП = 23, = 6, =1,32, =1,00,  =0,72,  = 2,08, =0, =0, =1 [2], следовательно, после подстановки значений в (5.8) и (5.9), получается:





Трудоемкость разработки рабочего проекта  зависит от функционального назначения ПП, количества разновидностей форм входной и выходной информации, сложности алгоритма функционирования, сложности контроля информации, степени использования готовых программных модулей, уровня алгоритмического языка про­граммирования и определяется по формуле (5.10):

, (5.10)

где *КК*– коэффициент учета сложности контроля информации; *КЯ* – коэффициент учета уровня используемого алгоритмического языка программирования; *КЗ* – коэффициент учета степени использования готовых программных модулей; *КИА* – коэффициент учета вида исполь­зуемой информации и сложности алгоритма ПП.

Значение коэффициента *КИА* определяется из выражения (5.11):

, (5.11)

где , ,  - коэффициенты учета сложности алгоритма ПП и вида используемой информации для переменной. нормативно-справочной информации и баз данных соответственно; ,  - норма времени, затраченного на разработку РП на алгоритмическом языке высокого уровня разработчиком постановки задач и программного обеспечения соответственно, чел.дн.

Для разработанного ПП *КК*=1,16, *КЯ*=1,00, *КЗ*=0,8, =1,00, =0,48, =0,40, =6, =38 [2] ,следовательно, после подстановки значений в (5.10) и (5.11), получается:





При разработке программной продукции стадии «Технический проект» и «Рабочий проект» объединены в стадию «Технорабочий проект», поэтому трудоемкость ее выполнения  опреде­ляется по формуле (5.12):

*,* (5.12)

Таким образом: **

Трудоемкость выполнения стадии «Внедрение» tВ рассчитывается по формуле (5.13):

*,* (5.13)

где ,  - норма времени, затрачиваемого разработчиком постановки задач и разработчиком ПО соответственно на выполнение процедур внедрения ПП, чел.дн.

Для разработанного ПП =8, =8 [2], следовательно, после подстановки значений в (5.12), получается:

**

В итоге, после подстановки полученных значений в (5.1), трудоемкость разработки ПП составляет:



Таким образом, трудоемкость составляет 143 дней.

* + 1. Определение цены программной продукции

Процесс разработки сложной программной продукции сопровождается, кроме решения чисто программных аспектов, необходимостью решения многих социальных и экономических проблем. Одна из серьезных экономических проблем - определение стоимости ПП.

Если ПП рассматривается и создается как продукция производственно-технического назначения, допускающая многократное тиражирование и отчуждение от непосредственных разработчиков, то ее цена определяется по формуле (5.14)

, (5.14)

где *С* - затраты на разработку программной продукции (сметная себестоимость); *К* - коэффициент учета затрат на изготовление опытного образца ПП как продукции производственно-технического назначения (*К* = 1,1...1,2); *Пр* - нормативная прибыль, рассчитываемая по формуле (5.15)

, (5.15)

где *рн* - норматив рентабельности, %; *См* - материальные затраты, руб./изд.

Затраты на разработку программной продукции могут быть представлены в виде сметы затрат, включающей в себя следующие статьи: материалы, специальное оборудование, основная заработная плата, дополнительная заработная плата, отчисления на социальное нужды, производственные командировки, накладные расходы, контрагентские расходы.

1) Материалы.

В статье указываются суммарные затраты на материалы, приобретаемые для разработки данной ПП.

Для разработки ПП приобретение материалов не требуется, то есть *См* = 0 руб.

2) Специальное оборудование.

Затраты, связанные с использованием вычислительной техники, определяют по формуле (5.16):

, (5.16)

где  - время использования ЭВМ для разработки данного ПП, ч; *КИ*- поправочный коэффициент учета времени использования ЭВМ;  - цена 1-го часа работ ЭВМ, руб.; *КЭ* - коэффициент учета быстродействия ЭВМ (*КЭ*=1,0 - быстродействие ЭВМ более 20\*1030 опер./с, *КЭ* =1,2 - быстродействие ЭВМ менее 20\*1030 опер./с).

Для разработанного ПП =44, *КИ* =1,19, =15 руб., *КЭ* =1,0 [2], следовательно, после подстановки значений в (5.16), получается:



3) Основная заработная плата.

В статью включается основная заработная плата всех исполнителей, непосредственно занятых разработкой данного ПП, с учетом их должностного оклада и времени участия в разработке. Расчет ведется по формуле (5.17):

, (5.17)

где *Зi*- среднемесячный оклад i-го исполнителя, руб.;  - среднее количество рабочих дней в месяце; *ti* - трудоемкость работ, чел.дн. (определяется из календарного плана-графика).

Для разработанного ПП должностной оклад программиста составляет Зi=10 тыс.руб., среднее количество рабочих дней в месяце =22, трудоемкость работ *t*=146,23, следовательно, после подстановки значений в (5.17), получается:



4) Дополнительная заработная плата.

В статье учитываются все выплаты непосредственным исполнителя за время (установленное законодательством), непроработанное на производстве, в том числе: оплата очередных отпусков, компенсации за недоиспользованный отпуск, оплата льготных часов подросткам и др. Расчет ведется по формуле (5.18):

, (5.18)

где  - коэффициент отчислений на дополнительную зарплату.

Для разработанного ПП =0,2 [2], следовательно, после подстановки значений в (5.18), получается:



5) Отчисления на социальное страхование.

В статье учитываются отчисления в единый социальный налог по установленному законодательством тарифу от суммы основной и дополнительной заработной платы (5.19):

, (5.19)

где - коэффициент отчислений на социальное страхование.

Для разработанного ПП [2], следовательно, после подстановки значений в (5.19), получается:



6) Накладные расходы.

В статье учитываются затраты на общехозяйственные расходы, непроизводительные расходы и расходы на управление. Накладные расходы определяют в процентном отношении к основной заработной плате (5.20):

, (5.20)

где  - коэффициент накладных расходов.

Для разработанного ПП [2], следовательно, после подстановки значений в (5.20), получается:



7) Производственные командировки.

В статье учитываются все расходы на командировки, связанные с разработкой данной Ш. В ходе разработки программной продукции командировок не было.

8) Контрагентские расходы.

В статье учитываются затраты на выполнение сторонними организациями работ, непосредственно связанных с разработкой данного ПП. Сторонние организации к разработке программной продукции не привлекались. Результаты расчета сметной стоимости ПП приведены в таблице 5.1.

Таблица 5.1 Сметная стоимость программного продукта

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Наименование статьи | Сметная  себе­стоимость, руб. | Удельный вес, % |
| 1. | Материалы | 0 |  |
| 2. | Специальное оборудование | 785,4 |  |
| 3. | Основная заработная плата | 66468,18 |  |
| 4. | Дополнительная заработная плата | 13293,64 |  |
| 5. | Отчисления на социальное страхование | 27119 |  |
| 6. | Накладные расходы | 33234,09 |  |
| 7. | Производственные командировки | 0 |  |
| 8. | Контрагентские расходы | 0 |  |
|  | Итого | 140900,31 |  |

Нормативная прибыль, рассчитываемая по формуле (5.15) составляет:



По формуле (5.14) определяется цена ПП:



* + 1. Капитальные вложения

Если еще не было программы (алгоритма) для решения данных задач, то разработанная программа сопоставляется с решением этих задач вручную, т.е. с тем, как они решались раньше. В этом случае дополнительные капитальные вложения Кд (р./одного потребителя программы), связанные с внедрением разработанной программы (алгоритма), определяются так:

*КД = (Тм.в КЭВМ* / *Тпол) + Цп* ,

где *КЭВМ* – капитальные вложения в ПЭВМ, для которой предназначена данная программа (алгоритм);

*Тпол* – полезный годовой фонд работы этой ПЭВМ (за вычетом простоев в ремонте), ч/год;

*Тм.в* – машинное время ПЭВМ, нужное данному потребителю для тех задач, которые он решает с помощью разработанной программы, машино-ч/год;

*Цп* – цена новой программы (алгоритма), которую должен приобрести потребитель, р./программу.

*КД* = (500 \* 20000 / 2112) + 198669,43 = 203404,28

* + 1. Эксплуатационные расходы

Расходы, связанные с эксплуатацией программы (алгоритма) *Е* (р./год на потребителя программы), определяются как

*Е* = (*Тм.в Цэвм*) + *Цп* / *Тс*,

где *Тм.в* – продолжительность машинного времени ПЭВМ, используемой в течение года для решения задач с помощью данной программы (алгоритма), машино-ч/год/потребителя алгоритма (программы);

*Цэвм* – эксплуатационные расходы, приходящиеся на 1 ч машинного времени этой ПЭВМ, р./машино-ч (по табл. 66 Цэвм = 5 [2]);

*Цп* – цена разработанной программы (алгоритма), р./программу;

*Тс* – срок службы данной программы (алгоритма).

Е = (500 \* 5) + 198669,43 / 10 = 42233,88

Если в прежнем варианте задача решалась вручную, то экономия эксплуатационных расходов *Еэ*, получаемая у одного потребителя данной новой программы (алгоритма), составит

*Еэ*= (1 + *αд*) (1 + *αcc*) *Зi* – *Е*,

где *Зi*  – основная заработная плата *i*-го работника, решавшего эту задачу вручную, р./год;

*Е* – эксплуатационные расходы.

*Еэ* = (1 + 0,2) \* (1 + 0,34) \* 6 \*15000– 42233,88= 102486,12

* + 1. Показатель эффективности и годового экономического эффекта

Срок окупаемости дополнительных капитальных вложений Ток в новом варианте по сравнению с прежним составит

Ток =  = 203404,28 / 102486,12≈ 1,98.

При Ток = 3 года < Тн = 6,6 года применение разработанной программы (алгоритма) является экономически эффективным.

В тех случаях, когда использование новой программы позволяет экономить как капитальные вложения (*К2* < *К1*), так и эксплуатационные расходы (*Е2* < *Е1*), годовой экономический эффект *Эг*, получаемый от внедрения новой программы (алгоритма), определяется как

*Эг* = *Еэ* + *ен Кэ* = 102486,12 + 0,15\*203404,28 = 132996,76.

Вывод по главе

В данной главе приведен бизнес-план реализации разработанного программного продукта, а также рассчитаны основные экономические показатели, свидетельствующие об экономической оправданности разработки. Кроме того, были подробно описаны достоинства и недостатки программ-конкурентов. Сводные экономические показатели по разработке и эксплуатации программы приведены в таблице 5.2.

Таблица 5.2. Сводные экономические показатели

|  |  |  |
| --- | --- | --- |
| Показатель | Размерность | Значение |
| Затраты на разработку программы | р. | 140900,31 |
| Цена программного продукта | р. | 198669,43 |
| Капитальные вложения | р./год | 203404,28 |
| Экономия эксплуатационных расходов | р./год | 102486,12 |
| Годовой экономический эффект | р./год | 132996,76 |
| Срок окупаемости, Ток | годы | 1,98 |

Вывод по главе

Заключение

Список литературы