Реферат посвящен теме криптографии – а именно криптоалгоритмам. Нужно рассмотреть алгоритмы шифрования и дешифровки, выявить минусы и плюсы каждого типа, выбрать несколько алгоритмов и реализовать их на Питоне.

Криптография как наука появилась в 16 веке. Позже, в 20 веке, с появлением ЭВМ стало возможно создавать шифры, практически неподдающиеся взлому человеком. Сегодня взлом хорошо зашифрованной информации даже на мощных компьютерах займет долгие годы.

Криптоалгоритмы делятся на 2 категории: симметричные и ассиметричные

Симметричные алгоритмы – самые «удобные». Они просты в реализации и хорошо изучены (так как появились раньше ассиметричных). Требуют одного ключа.

Ассиметричные – есть открытый (публичный) ключ шифрования и закрытый (личный). Открытый используется только для шифровки, закрытый – для дешифровки. Требуют длинных ключей и большой вычислительной мощности, но тяжелее поддаются взлому.

Помимо простой исследовательской работы я также реализую некоторые алгоритмы на языке Python, так как он одновременно мощный и достаточно прост в освоении. Будет использоваться среда PyCharm, для перевода программы в \*.exe я воспользуюсь интерпретатором.

Процесс работы я разделил на несколько этапов.

* Поиск алгоритмов шифрования
* Разбор принципа их работы
* Написание основной части реферата
* Написание выбранных алгоритмов на языке Python

Используемые источники:

* Википедия
* http://www.codenet.ru/progr/alg/enc/ для разбора типов алгоритмов
* <http://younglinux.info/python.php> для изучения Python

**Терминология**

**Взаимно простые числа**

Взаимно простые числа – несколько целых чисел, которые не имеют общих делителей, кроме ±1. К примеру, числа 14 и 25 — взаимно простые. 13, 39, 51 тоже являются простыми числами – между всеми тремя одного общего делителя нет.

**Функция Эйлера**

Функция Эйлера φ(*n*) – арифметическая функция, равная количеству натуральных чисел, меньше числа *n* и взаимно простых с ним. К примеру, φ(14)=6, так как **1**, 2, **3**, 4, **5**, 6, 7, 8, **9**, 10, **11**, 12, **13 –** всего 6 чисел, взаимно простых с 14. Для простого числа *n* функция Эйлера равна *n*-1, так как у простого числа нет делителей, кроме ±1и самого *n.*

**Первообразный корень**

Первообразный корень по модулю *m* – это целое число *g* такое, что

*g* φ(*m*)≡1 (mod *m*) и *gl* ≢ 1 (mod *m*) при 1≤ *l* < φ(*n*). В качестве примера возьмем число 3, которое является первообразным корнем от числа 7. Проведем для него проверку.

30≡1 (mod 7) *1 mod 7 = 1*

31≡3 (mod 7) *3 mod 7 = 3*

32≡2 (mod 7) *9 mod 7 = 2*

33≡6 (mod 7) *27 mod 7 = 6*

34≡4 ( mod 7) *81 mod 7 = 4*

35≡5 (mod 7) *243 mod 7 = 5*

36≡1 (mod 7) *729 mod 7 = 1*

**Односторонние функции**

 Односторонняя функция (или труднообратимая функция) – математическая функция, значение которой легко вычислить, но трудно подобрать аргумент по значению. Чем сложнее дается этот подбор, тем выше стойкость алгоритма, основанного на этой функции.

 В качестве примера возьмем функцию вида *Yx (mod P)* (нахождение остатка от деления *Yx*на число *Р*). Найти исходные значения данной функции можно только перебором. Для наглядности попробуем найти значение *x* функции 5*x*(mod 7) = 2.

* 51 mod 7 = 0
* 52 mod 7 = 25 mod 7 = 4
* 53 mod 7 = 125 mod 7 = 6
* 54 mod 7 = 625 mod 7 = 2 →*х*=4

У человека без калькулятора поиск аргумента может занять от 2 до 5 минут. В криптоалгоритмах для односторонних функций берутся числа порядка 10300 , что делает задачу невозможной ни для человека, ни для компьютера\*. Часто за *P* берут простое число, за *Y* – первообразный корень по модулю *Р*. Таким образом, вероятность получения значений от одного до *Р*-1 при разном *х* распределяется равномернее, что повышает стойкость алгоритма.

**Хеш функция**

Хеш функция – односторонняя функция, которая преобразует входящую последовательность переменного размера в выходную последовательность постоянного размера (в хэш). Применяются практически во всех криптоалгоритмах и на сайтах с регистрацией. Так, в базе данных сайта не хранятся сами пароли пользователей – хранятся хэши, которые проверяются в момент входа.

 Еще одно применение хэш функций – контрольные суммы, которые позволяют проверить целостность файла. К примеру, недавно был взломан сайт дистрибутива операционной системы Linux Mint и изменены файлы образа загрузочного диска. При смене частей файла сменился и его хэш (контрольная сумма), и это позволило заметить взлом.

**Электронная Цифровая Подпись**

**ЭЦП –** реквизит документа, получаемый в результате преобразования информации файла при помощи закрытого ключа. Служит для проверки отсутствия искажения информации файла и проверки авторства.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*Прим. автора: при увеличении длины ключа время взлома увеличивается до 7 раз. Для алгоритма RSA увеличение ключа на 1 бит увеличивает число переборов в 2 раза. Так ключ длиной в 256 потребовал бы 2256 переборов. Если написать это число, то оно не помещается на строке этого документа - это 115792089237316195423570985008687907853269984665640564039457584007913129639936 возможных ключей. Произнести это число сложнее, чем некоторые немецкие слова.

 Современные компьютеры (не берем в расчет квантовые) для завершения этой операции, при идеальных условиях, требовали бы энергию, равной энергии от взрыва сверхновой (с КПД равным 100%).

 К примеру, настольный процессор Intel последнего поколения с архитектурой в 14 нм имеет тактовую частоту 4ГГц или 4 миллиарда операций в секунду. Если принять, что 1 ключ – 1 операция, то мы получаем 28948022309329048855892746252171976963317496166410141009864396001978 секунд.



 Разделив это число на число секунд в году, получим

917935765770200686703854206372779584072726286352427099501027 лет.

 Для сравнения – нашей вселенной, по приблизительным подсчетам, около 14 миллиардов лет.

**Глава 1**

**Глава 1. Симметричные алгоритмы.**

**Определение**

Симметричные алгоритмы – алгоритмы, использующие один ключ для шифровки и расшифровки. Такие алгоритмы появились еще в далекой древности, и в их основе лежала простая перестановка.

 Главная проблема таких алгоритмов – передача ключа. Ключ должен находиться у обеих сторон, иначе зашифрованный текст теряет смысл. Данная проблема решается с использованием односторонних функций.

**Передача ключа по открытому каналу**

Для простоты дальнейшего понимания попробуем сначала решить задачу: как девушке Алисе переслать другу Бобу секретное письмо, если все письма вскрываются и читаются Евой?

В реальности у этой задачи очень простое решение. Сначала Алиса кладет письмо в сейф и запирает его на замок, после чего отправляет сейф с вложенным письмом Бобу. Боб, получив сейф, надевает параллельно свой замок и отправляет посылку обратно Алисе. Алиса, получив сейф, снимает с него свой замок и отправляет опять Бобу. Боб, получив сейф, отпирает его и достает письмо. Задача решена.

Но для криптографии подобное решение не годится – шифры должны «сниматься» в строго обратном порядке.

Разберем другой пример, в котором используется одностороннюю функцию. Ключ у нас будет представлять собой цвет, а роль односторонней функции сыграет смешивание краски.

1 шаг: Алиса и Боб договариваются о двух цветах, которые не являются секретными. Пускай это будут желтый и белый цвет. Затем они индивидуально смешивают эти цвета.

Желтый + Белый = Бледно-желтый.

2 шаг: Алиса и Боб примешивают свои секретные цвета к полученной смеси. Секретный цвет Алисы – синий, секретный цвет Боба – красный.

Синий + Бледно-желтый = Бледно-зеленый

Красный + Бледно-желтый = Бледно-оранжевый

3 шаг: Алиса и Боб обмениваются по почте полученными смесями и примешивают к ним свои цвета. В результате у них обоих получается бледно-коричневый цвет.

Синий + Бледно-оранжевый = Бледно-коричневый

Красный + Бледно-зеленый = Бледно-коричневый

Ева не может получить этот цвет, так как ей нужно знать секретные цвета. Для этого ей понадобилось бы размешать краску обратно, что невозможно.

Теперь разобрать пример решения данной задачи при помощи математической односторонней функции.

1 шаг: Алиса и Боб договариваются о значениях для односторонней функции. Значения не являются секретными. Допустим, что эти значения 9 и 14. Функция будет иметь вид 9х (mod 14)

2 шаг: Алиса и Боб выбирают свои случайные числа, которые хранят в секрете. Предположим, что Алиса выбрала число 5, а Боб – 7. Обозначим эти числа за А и В соответственно.

3 шаг: Алиса и Боб подставляют свои секретные числа в функцию и вычисляют значение. 95 (mod 14) = 59 049 mod 14 = 11. 97 (mod 14) = 4782969 (mod 14) = 9. Эти числа обозначим за С и Е соответственно.

4 шаг: Алиса и Боб обмениваются вычисленными значениями и вычисляют значение функции со своим и полученным числом в качестве аргумента, т.е. Алиса вычисляет ЕА (mod 14), Боб, соответственно, вычисляет СВ (mod 14). 95 (mod 14) = 59 049 mod 14 = **11**. 117 (mod 14)= 19487171 mod 14 = **11**. Полученное в результате всех операций число 11 будет секретным ключом для шифрования передаваемых данных.

Описанное выше решение произведено по алгоритму Диффи-Хеллмана, который был представлен в 1976 году двумя криптографами и произвел настоящую революцию в мире шифрования. Сегодня этот алгоритм является основополагающим в протоколах передачи данных: VPN, HTTPS и других.

**Глава 2**

**Глава 2. Асимметричные алгоритмы.**

**Определение**

**** Асимметричные алгоритмы – алгоритмы, использующие два ключа – для шифрования и дешифрования. Важно, что ключ для шифрования является открытым, для дешифрования, соответственно, закрытым (секретным).

 Асимметричные алгоритмы требуют более длинных ключей и в целом работают медленнее, чем симметричные алгоритмы, но часто применяются при необходимости распределения большого количества ключей между пользователями (т.к. не надо каждый раз заново генерировать ключи).

 Алгоритм можно весьма просто описать, если использовать ситуацию с Бобом и Алисой. В нашем случае Алиса делает один ключ, который является строго секретным и не должен передаваться, и много замков, подходящих под этот ключ. Любой человек может использовать замок для закрытия сейфа с посылкой (зашифровать), но не может снять этот замок (расшифровать), так как ключ находится у Алисы.

**Генерация ключей для ассиметричного шифрования**

Для генерации ключей, как и в случае с симметричными алгоритмами, односторонние функции, используются односторонние функции.

 Разберем пример передачи данных по упрощенному алгоритму асимметричного шифрования RSA.

1. Выбираем два неравных простых числа p и q.
2. Вычисляем модуль N=p\*q.
3. Вычисляем значение функции Эйлера от N. Для простого числа *a* функция Эйлера равна $φ$(*a*)=*a*-1, следовательно, для модуля N функция равна $φ$(N)=(p-1)(q-1).
4. Для формирования ключа нам нужны еще два числа – *e* и *d*. Число *e* должно лежать в интервале 1< *e* <$φ\left(N\right)$ и быть взаимно простым со значением $φ\left(N\right)$. Число *d* вычисляется с использованием формулы *d\*e=1(mod* $φ$*(N)).* Эту формулу можно также записать в виде *d\*e mod* $φ$*(N)=1.*
5. Мы получили две пары, которые являются ключами. Пара (*e,N*) – открытый ключ и пара (*d,N*) – закрытый ключ.

**Процесс шифрования и обмена данными по упрощенному алгоритму RSA**

Боб шифрует сообщение *m* ключом Алисы по формуле *memod(N)*. Алиса, получив зашифрованное сообщение *с*,расшифровывает его через формулу *m=cdmod(N)*

Пример:

1. За два простых числа возьмем 5 и 13.
2. Модуль N=5\*13=65
3. Функция Эйлера для N: 4\*12=48
4. Число *е* – 31. Число *d –* 79. 31\*79 mod 48 = 2449 mod 48 = 1

Перейдем к шифрованию. Для примера возьмем слово “RSA”. Переведем его в цифровой код, используя пронумерованный алфавит

 18 19 1

 R S A

Теперь нам необходимо провести операцию шифрования над каждым числом. Для удобства мы воспользуемся программой на Python.

out1=[]; #Пустой массив для хранения зашифрованного сообщения

out2=[]; #Пустой массив для хранения дешифрованного сообщения

inp=str(raw\_input('Enter message: ')); #ввод текста, преобразованного в код

user=inp.split(); #преобразование текста в массив

n=**65**;

e=**31**;

d=**79**;#числа-ключи

c=**0**;#буфер, промежуточная перемена

**for** i **in** range(**0**, len(user)):

 user[i]=long(user[i]); #конвертируем из string в long int

**for** i **in** range(**0**,len(user)): #от нуля до конца массива

 c=user[i]\*\*e;

 c=c%n

 out1.append(c);

**for** i **in** out1:

 **print** i,;

**print**;

**for** i **in** range(**0**, len(out1)):

 c=out1[i]\*\*d;

 c=c%n;

 out2.append(c);

**for** i **in** out2:

 **print** i,;

**print**;

Результат:

Enter message: 18 19 1

47 59 1

18 19 1

 Важно понимать, что описанный выше алгоритм не применяется в таком виде, так как местами может быть легко «взломан». К примеру, взломщик может сам сгенерировать и зашифровать простые ответы (да, нет, возможно и т.д.), а затем перехватывать сообщение и сверять его со своим. Это решается использованием соли.

 **Соль** (или модификатор)в криптографии – строка случайных данных, которая подается на вход функции шифрования вместе с входными данными. По понятным причинам, соль должна быть уникальной для каждого входного значения. Процесс добавления соли к данным называется солением.

 Упрощенный пример генерации соли на языке Python с использованием модуля random

**from** random **import** randint

**from** string **import** ascii\_lowercase

f=list(ascii\_lowercase)

salt=[]

**for** i **in** range(0,26):

    c=randint(0,1)

    **if** c==0:

        x=randint(0,25)

        salt.append(f[x])

    **else**:

        salt.append(x)

salt1=''.join([str(i) **for** i **in** salt])

**print**(salt1)

**Глава 3**

**Глава 3. Примеры симметричных/ассиметричных алгоритмов**

**Симметричные алгоритмы**

**DES**

DES(Data Encryption Standard – стандарт шифрования данных) – алгоритм, разработанный и утвержденный в 77 году. Имеет длину ключа в 56 бит и длину блока (части шифруемой информации) в 64 бита. Довольно долгое время был основным алгоритмов, но в 98 году был взломан суперкомпьютером менее чем за три дня. Учитывая закон Мура, согласно которому производительность микросхем увеличивается в два раза каждые 18 месяцев, алгоритм был признан ненадежным и заменен на 3DES – грубо говоря, алгоритм с утроенным количеством шифрований. Ожидаемо, 3DES в три раза медленнее DES, но его взлом занимает в миллиарды раз больше времени. Сегодня этот алгоритм используется в обработке кредитных кард VISA.

**IDEA**

IDEA (International Data Encryption Algorithm – Международный Алгоритм Шифрования Информации) – алгоритм, разработанный в 91 году. Использует 128 битный ключ и блок в 64 бита. Если в блоке не хватает информации, его дополняют до 64 бит. Во время шифрования каждый блок разбивается на 16 бит и шифруется разными способами.

 Основные операции, используемые алгоритмом:

-сложение по модулю 216

-умножение по модулю 216+1

-логическая операция XOR (побитовое отрицание)

Эти операции взяты из разных разделов алгебры и не удовлетворяют двум законам: распределительному и сочетательному. Их применение увеличивает криптостойкость алгоритма при использовании меньшего количества времени.

**Blowfish**

Blowfish – свободно распространяемый криптографический алгоритм, написанный в 93 году. Имеет переменную длину ключа, от 32 до 448 бит. Алгоритм выполнен на простых операциях: XOR, подстановка и сложение. Благодаря этому алгоритм Blowfish является надежным алгоритмом, что уменьшает вероятность и ошибки при шифровании, и не требует большего количества ресурсов: для его работы достаточно около 5Кб памяти.

Blowfish широко используется в обеспечении сетевой безопасности: в сетевом протоколе SSH(Secure Shell) и при создании зашифрованных каналов передачи данных, к примеру, VPN.

**Ассиметричные алгоритмы**

**RSA-256**

Алгоритм, описанный в предыдущей главе

**DSA**

DSA – алгоритм, опубликованный в 94 году и используемый для создания цифровой подписи, которая позволяет проверить целостность передаваемого файла. Подписать файл может только один человек, использую закрытый ключ, а проверить подпись может каждый, используя открытый ключ. Надежность ключа обеспечивается сложностью дискретного логарифмирования (обращения односторонней функции).

**Rabin**

Криптосистема Рабина – система с открытым ключом, по безопасности не уступающая RSA. Данная система полагается на сложность поиска квадратных корней по модулю составного числа.

 Главное неудобство при использовании криптосистемы Рабина – побочные сообщения. При шифровке одного сообщения получаются еще 3 побочных. Если зашифрованное сообщение является текстовым файлом, то трудностей с поиском исходного сообщения не возникает, но при пересылке случайного потока чисел (для создания ЭЦП, как пример) задача становится невозможной

**Глава 4**

**Криптоалгоритмы с Python**

 Вся реализация сильно упрощена для понимания читателей и самого автора

Для начала оптимизируем код по генерации соли из предыдущей главы. В этом нам поможет одна из возможностей Питона – быстрая генерация массивов/списков.

К примеру, массив, заполненный четными квадратами чисел от 0 до 10 можно получить двумя способами.

Способ 1. С использованием цикла и условного оператора.

1. x=[]
2. **for** i **in** range(1,11):
3. **if** i\*\*2 % 2 == 0:
4. x.append(i\*\*2)
5. **print** x

Вывод программы: [4, 16, 36, 64, 100]

Сначала мы создаем пустой массив, затем в цикле считаем квадраты чисел и проверяем их на четность. Если условие выполняется, то мы добавляем квадрат в массив.

Упростим программу, используя генератор списков.

1. x=[i\*\*2 **for** i **in** range(1,11) **if** i % 2==0]
2. **print** x

Вывод программы: [4, 16, 36, 64, 100]

 Как видно, программа выдает тот же результат при меньшем использовании места и памяти.

 Теперь применим это к программе для генерации соли, которую мы использовали в главе 2. Для генерации цифры или буквы применим функцию, которая будет возвращать нужное значение. Также мы уменьшим соль в два раза – иначе ее добавление может сильно затруднить понимание сообщения.

**from** string **import** ascii\_lowercase

**from** random **import** randint

a=list(ascii\_lowercase)

**def** returnsalt():

    **def** lon():

        c=randint(0,1)

        **if** c==0:

            **return** randint(1,26)

        **else**:

            **return** a[randint(0,25)]

    salt=[lon() **for** i **in** range(13)]

    salt = ''.join(str(v) **for** v **in** salt)

    **return** salt

Итак, теперь можно перейти непосредственно к шифрованию.

Возьмем гипотетическую ситуацию. Алиса и Боб – работники в одной компании, которая занимает небоскреб. В небоскребе присутствует локальная сеть, но все сообщения, которые пересылаются через нее, читаются начальством. При этом Алиса и Боб работают на разных этажах (7 и 39, к примеру), поэтому вариант личного общения напрямую тут не подходит

Поэтому Алиса и Боб могут прибегнуть к шифрованию. В примере мы это сделаем с использованием модуля RSA и записью в файлы. Файлы будут расположены на сервере компании, так что проблем с доступом ни у Алисы, ни у Боба не будет.

Итак, сначала Алиса создает на хранилище текстовый файл, и указывают к нему путь доступа. После этого она по описанному во второй главе алгоритму генерирует два ключа для шифрования и записывает один из этих ключей в файл для Боба. Боб получает ключ, с его помощью шифрует сообщение и сохраняет его в файл. Алиса обновляет файл, дешифрует и т.д.

Начнем.

Алисе необходимо сгенерировать пару ключей и записать открытый ключ в текстовый файл.

**import** rsa

pubkey, privkey=rsa.newkeys(512)

of=open("public\_key\_fmalice.txt", "rw+")

of.write(str(pubkey))

of.close()

Бобу необходимо считать публичный ключ, зашифровать с его помощью сообщение, а затем записать его в файл.

**import** rsa, salt

f1=open("public\_key\_fmalice.txt", "r+")

key=f1.read()

message=b"Hey there! Do you know the latest news?"+ salt.returnsalt()

cm = rsa.encrypt(message, key)

f2=open("bob\_message.txt", "r+")

f2.write(cm)

f2.close

 Наконец, Алиса считывает сообщение и выводит его у себя.

received\_message=""

f5=open("bob\_message.txt", "rw+")

**for** line **in** f5:

    received\_message+=line

message=rsa.decrypt(received\_message, )

**print**(message, privkey)

Как итог: RSA легко реализуется на Питоне и вполне может быть применен дома/в школе/на работе.