ОТЧЕТ

Шифр Twine

# Общее описание алгоритма

Тип – блочный

Длина ключа – 80, 128 бит

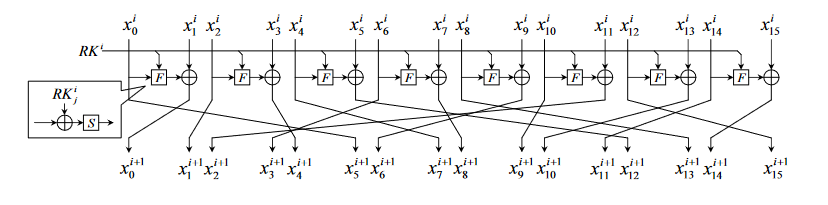
Длина блока – 64 бита

Twine изначально разрабатывался как шифр для слабого железа или для микроконтроллеров. Если в дальнейшем посмотреть на рисунок, где показана раундовая функция, то станет видно, что там все действия в ней производятся, в основном, с блоками по 4 бита.

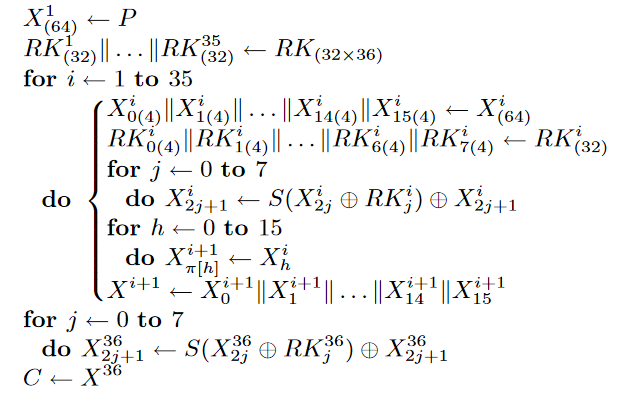
# Описание криптографических свойств алгоритма и принципов его построения

Шифр Twine это 64-битный блочный шифр, что поддерживает 2 длины ключа – 80 и 128 бит. Получая на вход 64 бита открытого текста – P, раундовый ключ – RK (который получается из основного ключа – K), он генерирует шифртекст – C. Чтобы превратить K в RK используется функция, которая называется – keySchedule. Раундовая функция содержит в себе нелинейный слой, использующий 4 битный Sbox, и слой рассеивания с перестановкой в 16 блоков. Сама раундовая функция повторяется 36 раз, на последней итерации перестановка не используется.

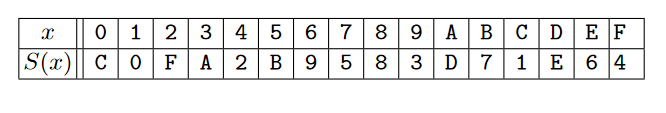
На словах не совсем понятно как работает шифр, поэтому лучше будет сперва проиллюстрировать его раундовую функцию, а затем функцию keySchedule.



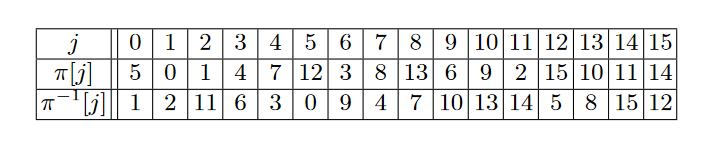
На псевдокоде алгоритм выглядит так –



Sbox –



Перестановка –



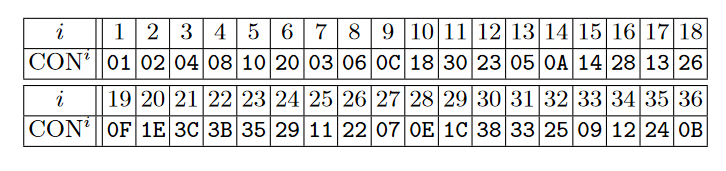
Как видно по алгоритму на псевдокоде – раундовый ключ RK представляет собой конкатенацию из 32 битных раундовых ключей 36 раз.

В самой раундовой функции очень странная постановка 4 битных блоков после преобразования. Позиция, на которую попадет каждый блок, определяется в таблице перестановок. То есть, наш нулевой блок перейдет на пятую позицию, а первый в нулевую и так далее.

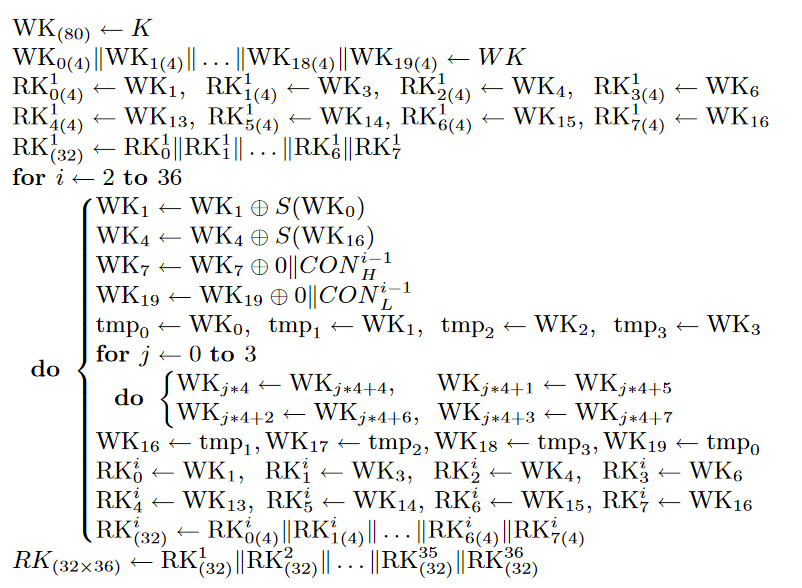
В самой же раундовой функции берется первый раундовый 32 битный ключ, затем открытый текст разбивается на 16 4-битных блока, раундовый ключ разбивается на 8 4-битных блока. Затем результат xor между блок блоками теста и ключа закидывается в Sbox и уже этот результат побитово складывается с последующим блоком открытого теста. Затем эти результаты занимают позиции согласно перестановке и происходит конкатенация в одну строку. Ну и на последней, 36, итерации не происходит разбиение на перестановку.

Теперь стоит рассмотреть процедуру получения раундового ключа – keySchedule. Для 80 битного и 128 битного ключа эти функции будут совсем незначительно отличаться. Сперва рассмотрим для 80 битного.

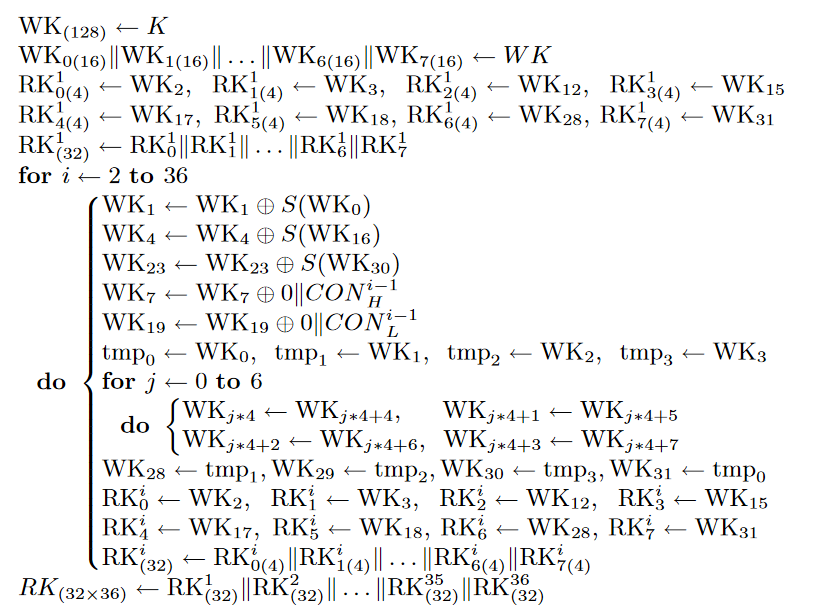
Раундовые константы –



Процедура для 80 битного ключа –



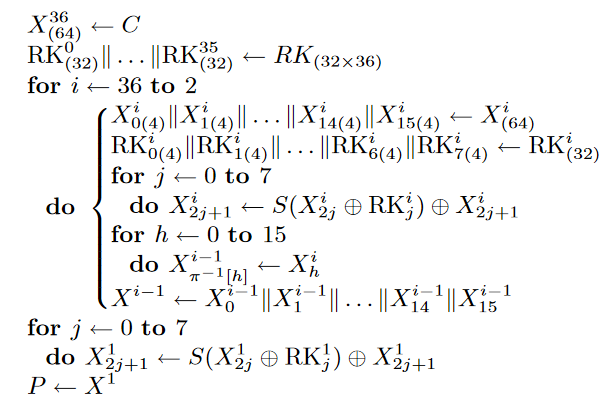
Для 128 битного –



На первый взгляд эта функция кажется сложной и перегруженной, но при этом она перегружена только в индексах блоков, которые берутся для дальнейших преобразований. В основном здесь то же самое, что мы рассматривали до этого, - то есть разбиение на 4 битные блоки и их последующее преобразование. Константы CONH и CONL получаются путем отбрасывания правых или левых 3 бит соответственно.

По итогу у нас выйдет конкатенация из 36 32-битных блоков ключей.

Процедура расшифрования, как и в большинстве, блочных шифров почти похожа на процедуру шифрования.



Как здесь видно – начинаются итерации с 36 индекса и доходят до 2, так как в самом конце не используется перестановка.

# Длина блока, ключа, количество раундов

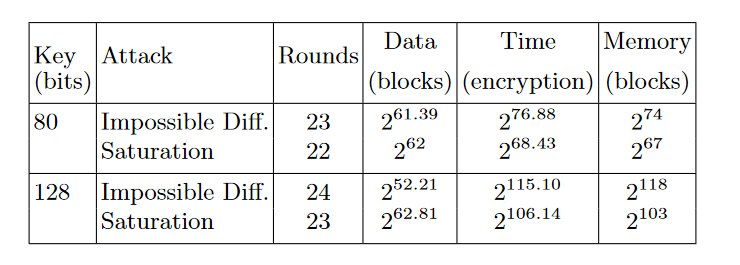
Указанно выше

# Максимальная теоретическая стойкость для тотального перебора

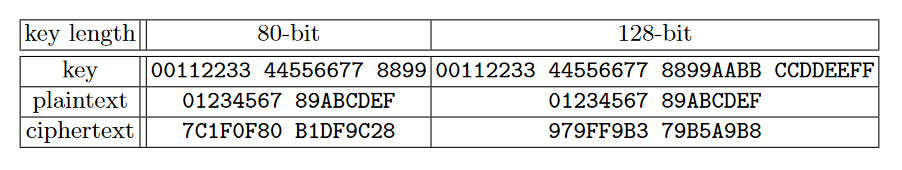
# Известные оценки стойкости по открытым источникам

От разработчиков приводятся атака на 23 и 24 раундовые вариации шифра с 80 и 128 битным ключом соответственно. На таком количестве раундов можно вскрыть шифр с помощью невозможной дифференциальной атаки (IDC). В связи с этим в настоящих реализациях используются 36 раундов. Также шифр с раундами меньше, чем 24 и 80 битным ключом уязвим к, так называемой, атакой насыщением (Saturation Attack).

Таблица для IDC и Saturation атак –



В качестве контрольного примера предлагаются следующие варианты –



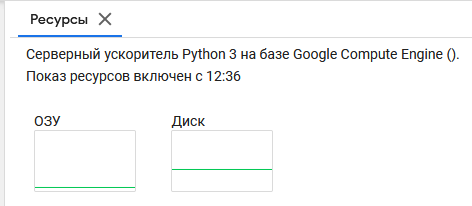
# Тестирование и описание характеристик реализации алгоритма

# Общее описание программной реализации алгоритма

Реализация на языке Python, среда выполнения – Google Colab. Сам Colab подключается к серверному ускорителю Python на базе Google Computer Engine.

ОЗУ (без запуска программы) – 0.74/12.69 Гб

Диск (без запуска программы) – 38,63/107,72 Гб



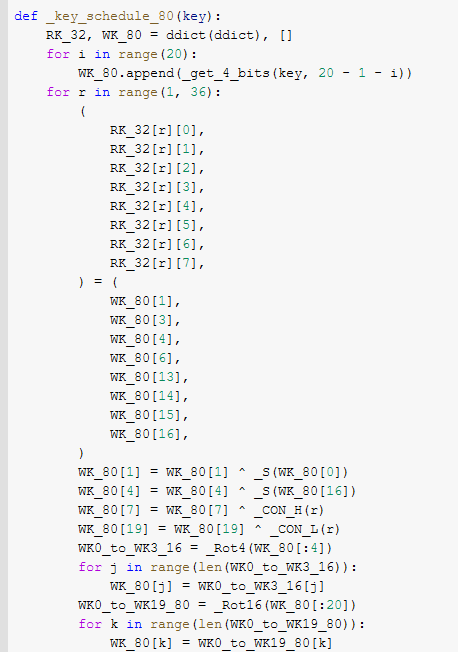
Больше доступных характеристик Colab просмотреть не дает.

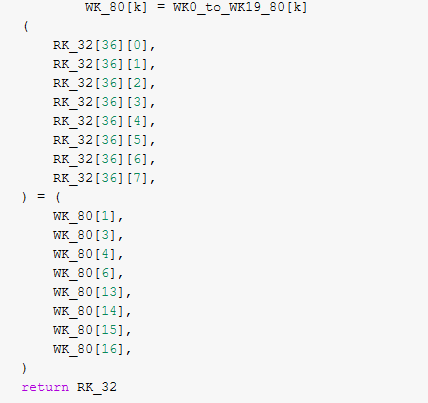
Есть возможность подключить аппаратный ускоритель расчетов, но это не для нашего случая, так как там на выбор есть GPU и TPU. Эти два ускорителя, в основном, используются для вычислений в нейронных сетях.

Реализация в отдельном файле на Github.

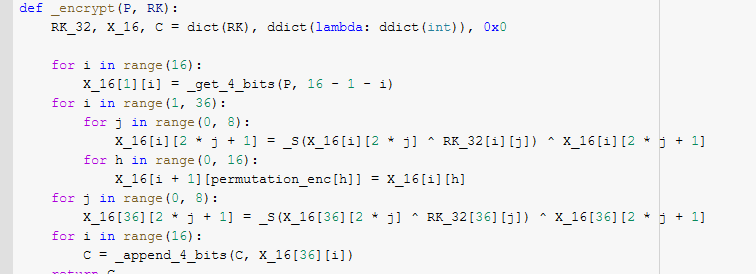
В качестве Sbox, раундовых констант и перестановки используются словари.

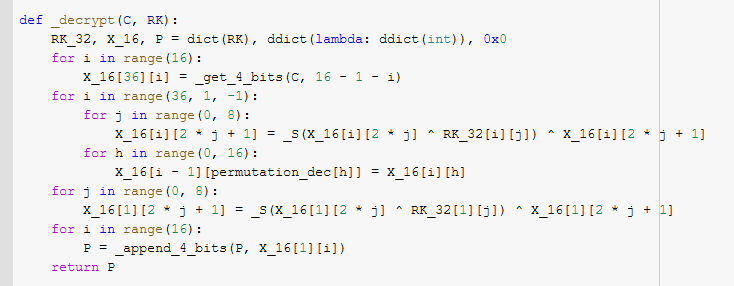
Для создания раундового ключа разной длины есть 2 функции – keySchedule80 и keySchedule128.





Шифрование происходит с помощью 2 функций - \_encrypt и encrypt. Сперва открытый текст подается на вторую функцию, там он разбивается на блоки и передается на вход первой функции.

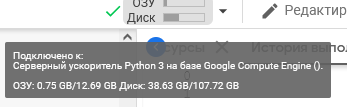


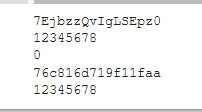


Расшифрование происходит по такой же схеме

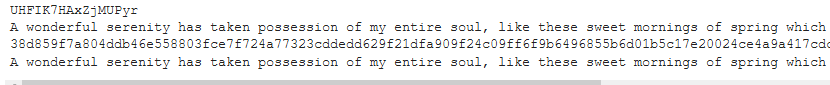
# Скорость выполнения тестового задания

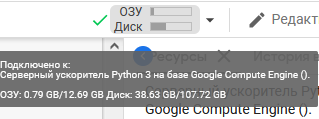
1 блок –

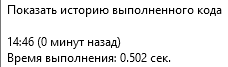




103 блоков –

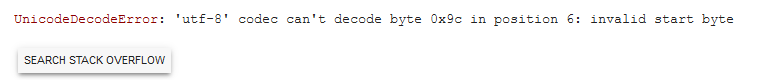






106 –

На данной итерации теста начинаются проблемы, так как декодирование сталкивается с ошибкой –



Это связано с тем, что символ, который подается на вход к декодеру не является символом из его кодировки UTF-8. Если же убрать декодер, то будет следующее –

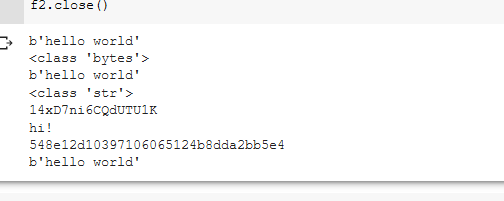


А это уже происходит из-за того, что на вход подается строка нечетной длины.

Решение данной проблемы – тотальный рефакторинг кода, а именно функций генерации раундовых ключей, шифрования и дешифрования.

Проверим, как он работает с файлами

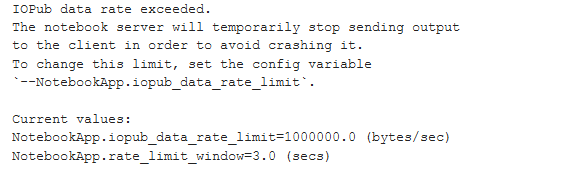
Произвольный файл –





Файлы большим размером очень долго обрабатываются в данной работе и в работе с хэшем.

Сперва скидывает данную ошибку –



Это значит, что в дальнейшем исполнение может завершиться само по себе. При этом функция шифрования продолжает выполняться.

В последующей работе данного бага не было, но время выполнения на файле в 1 мб было около 3,30 минут и, как не трудно почитать, на 100 мб понадобится 5,5 часов выполнения.





# Описание тестового стенда

Тестовый стенд и его конфигурация описана выше

# Фиксация нагрузки на ОЗУ и ЦП во время выполнения тестового задания

Фиксация была проведена выше

# Проверка встречной работы с таким же алгоритмом реализованным другим автором. Приоритетно выбирать алгоритмы, реализованные другими студентами этого курса

Не имеется возможности сделать проверку с другими студентами курса