Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сыктывкарский государственный университет имени Питирима Сорокина»

Институт точных наук и информационных технологий

Кафедра информационной безопасности

Допустить к защите

Зав. кафедрой, к.ф.-м.н., доцент

\_\_\_\_\_\_\_\_\_\_\_\_\_Л.С. Носов

«\_\_\_\_» июня 2015 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

Разработка криптографической библиотеки поточного шифрования на основе проекта eStream

Направление 090900 – «Информационная безопасность»

Научный руководитель:

Доцент к.ф.-м.н. \_\_\_\_\_\_\_\_\_ Ю.В. Гольчевский

«\_\_\_» июня 2015 г.

Исполнитель:

Студент группы 143 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р.А. Гашин

«\_\_\_» июня 2015 г.

Сыктывкар 2015

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc420150756)

[ГЛАВА 1. ПРОЕКТ eStream 5](#_Toc420150757)

[1.1 Проект «NESSIE» 5](#_Toc420150758)

[1.2 Проект eStream 6](#_Toc420150759)

[1.3 Алгоритм шифрования Salsa20 8](#_Toc420150760)

[1.4 Алгоритм шифрования Rabbit 11](#_Toc420150761)

[1.5 Алгоритм шифрования HC-128 14](#_Toc420150762)

[1.6 Алгоритм шифрования Sosemanuk 16](#_Toc420150763)

[1.7 Алгоритм шифрования Grain 19](#_Toc420150764)

[1.8 Алгоритм шифрования Trivium 20](#_Toc420150765)

[1.9 Алгоритм шифрования Mickey 22](#_Toc420150766)

[1.10 Общая схема функции шифрования/расшифровывания 24](#_Toc420150767)

[1.11 Область применения проекта eStream 25](#_Toc420150768)

[1.12 Безопасность алгоритмов проекта eStream 26](#_Toc420150769)

[ГЛАВА 2. БИБЛИОТЕКА estream.h 27](#_Toc420150770)

[2.1 Структура библиотеки estream.h 28](#_Toc420150771)

[2.2 Механизмы модернизации 32](#_Toc420150772)

[2.2.1 Использование макросов 33](#_Toc420150773)

[2.2.2 Использование битовых операций 34](#_Toc420150774)

[2.3 Модернизация алгоритмов 35](#_Toc420150775)

[2.4 Сравнение библиотеки estream.h и алгоритмов разработчиков 38](#_Toc420150776)

[ГЛАВА 3. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ БИБЛИОТЕКИ estream.h 43](#_Toc420150777)

[3.1 Рекомендации разработчикам 43](#_Toc420150778)

[3.2 Программа для шифрования файлов 44](#_Toc420150779)

# ВВЕДЕНИЕ

Бурное развитие информационных технологий привело к созданию глобальных компьютерных сетей общего пользования, доступ к которым открыт для всех пользователей по всему миру. По этим вычислительным сетям ежедневно передаются огромные объемы информации, в том числе и ограниченного распространения. Такая информация нуждается в обеспечении конфиденциальности при передаче. Самым надежным способом защиты по праву считается использование криптографических алгоритмов, основанных на математических преобразованиях.

Для защиты конфиденциальной информации, передающейся по вычислительным сетям, самыми эффективными признаны потоковые криптографические алгоритмы. Достоинством потоковых шифров является высокая скорость шифрования, и возможность работы с неограниченными потоками данных в режиме реального времени.

Передовые разработки в области потоковых шифров ведутся в России, США, Европе и Японии. Основой потоковой криптографии в США является алгоритм AES (Advanced Encryption Standard), работающий в режиме «счетчика». Эффективных атак на алгоритм, которые существенно бы понизили его стойкость, на сегодняшний день не выявлено. В результате чего AES признан одним из самых надежных алгоритмов в мире.

На территории стран Европейского Союза самым распространенным и надежным потоковым шифром считается алгоритм A5, который используется в европейской системе мобильной цифровой связи GSM (Group Special Mobile). Со времени публикации официальной документации (1994 год) в алгоритме были обнаружены некоторые недостатки, понижающие его надежность.

В феврале 2003 года в Японии завершился крупный проект по разработке криптографических алгоритмов «CRYPTREC». Его победителями стали 3 потоковых криптографических шифра MUGI, MULTI-S01 и 128-bit RC, предназначенные для использования в системе электронного правительства Японии. Недостатков и эффективных атак на алгоритмы не было выявлено, что позволяет говорить об их надежности и стойкости.

В России основная часть разработок в области потоковой криптографии засекречена, так как в большинстве случаев они используются в военных проектах для обеспечения связи.

Таким образом, обеспечение конфиденциальности информации, передаваемой по вычислительным сетям, с помощью криптографических алгоритмов является важным звеном в обеспечении информационной безопасности.

**Объектом исследования** данной выпускной квалификационной работы является обеспечение безопасности передаваемой информации по компьютерным сетям с помощью потоковых криптографических алгоритмов, а **предметом исследования** – потоковая криптографическая библиотека, разработанная на основе проекта eStream.

Цель выпускной квалификационной работы: разработка потоковой криптографической библиотеки на основе проекта eStream.

Для достижения цели выдвигаются следующие задачи:

1. изучение алгоритмов проекта eStream;
2. поиск и реализация возможностей для модернизации алгоритмов проекта eStream и сравнение новых реализаций с существующими;
3. разработка потоковой криптографической библиотеки estream.h;
4. разработка программных приложений для демонстрации возможностей разработанной библиотеки estream.h.

# ГЛАВА 1. ПРОЕКТ eStream

## 1.1 Проект «NESSIE»

Проект «NESSIE» (англ. New European Schemes for Signatures, Integrity and Ecryptions – новые европейские алгоритмы для электронной подписи, целостности и шифрования) – это научно-исследовательский проект для определения новых шифровальных алгоритмов, на базе которых должны быть созданы новые криптографические стандарты Европы. Конкурс проходил с января 2000 года по февраль 2003 года. Участниками проекта могли стать любые организации и частные лица, приславшие свой криптографический алгоритм для анализа.

Проект насчитывал 5 категорий шифров:

1. блочное шифрование данных;
2. ассиметричное шифрование
3. хэширование;
4. электронная подпись;
5. потоковое шифрование.

Отбор оптимальных алгоритмов, которые претендовали на победу, проходил в 3 этапа:

1. первичное изучение алгоритмов;
2. криптоаналитические исследования алгоритмов;
3. сравнение алгоритмов по нескольким критериям: быстродействие, минимальные требования к вычислительным ресурсам, производительность, гибкость и другие.

Всего на конкурс было заявлено 42 криптографических шифра, из которых 18 выбыло после 1-го этапа. В результате работы комиссии на 2-м и 3-м этапах победителями были объявлены 12 алгоритмов.

Победителями в соответствующих категориях стали:

1. блочные шифры: MISTY 1(Mitsubishi Electric Corp., Япония), Camellia (Nippon Telegraph and Telephone Corp., Япония и Mitsubishi Electric Corp., Япония), SHACAL-2 (Gemplus, Франция), AES (В. Рэймен и Й. Даймен, Бельгия);
2. ассиметричные шифры: ACE Encrypt (научно-исследовательская лаборатория IBM, Швейцария), PSEC-KEM (Nippon Telegraph and Telephone Corp., Япония);
3. хэш-функции: Two-Track-MAC (Левенский католический университет, Бельгия и Debis AG, Германия), UMAC (Intel Corp., США, университет Невады, США, научно-исследовательская лаборатория IBM, США, Technion, Израиль и университет Калифорнии в Дэвисе, США), Whirlpool (Scopus Tecnologia S.A., Бразилия и Левенский католический университет, Бельгия);
4. алгоритмы для электронной подписи: ECD SA (Certicom Corp., США и Certicom Corp., Канада), RSA-PSS (лаборатория RSA, США), SFLASH (Schlumberger, Франция).

В категории потоковое шифрование было представлено 6 алгоритмов, которые не прошли все испытания и были признаны непригодными для использования.

Данный факт послужил началом для старта проекта по выявлению новых потоковых криптографических алгоритмов eStream.

## 1.2 Проект eStream

Проект eStream (англ. European Stream – Европейский поток) – научно-исследовательский проект по выявлению новых потоковых криптографических алгоритмов. Был начат в феврале 2004 года, после взлома всех 6-ти шифров проекта «NESSIE». Официально проект был завершен в мае 2008 года. Последняя публикация документации шифров-победителей датируется январем 2012 года. После чего изменения в криптографические алгоритмы не вносились.

Созданию проекта eStream послужили результаты конференции, проходившей 14-15 октября 2004 года в городе Брюгге, «SASC – The State of the Art of Stream Ciphers» (Современное состояние потоковых шифров). Организаторы и слушатели конференции пришли к выводу, что необходимо организовать новый конкурс по выявлению потоковых криптографических алгоритмов.

В ноябре 2004 года началась приемная компания претендентов на участие в конкурсе. К участникам предъявлялись следующие основные требования: длина ключа максимум 128 бит; алгоритмы должны были работать быстрее, чем американский стандарт шифрования AES-128 (Advanced Encryption Standard) в режиме счетчика.

Все конкурсанты разделялись на 2 категории: «Программно-ориентированные алгоритмы» и «аппаратно-ориентированные алгоритмы».

В результате предварительного отбора, для участия в конкурсе были отобраны следующие участники: Frogbit, Fubuki, MAG, Mir-1, SSS, TRBDK3 YAEA, Phelix, Py, ABC, Achterbahn, DICING, Hermes8, NLS, Polar Bear, Pomaranch, SFINKS, TSC-3, VEST, WG, Yamb, ZK-Crypt, CryptMP, DECIM, Dragon, Edon80, LEX, MOSQUITO, Grain, HC-128, MICKEY, Rabbit, Salsa20, SOSEMANUK, Trivium, F-FC-SR.

На 1-м этапе проекта (29 апреля 2005 – 27 марта 2006) все участники подверглись общему анализу. Шифры были исследованы на надежность, производительность, корректность работы, простоту и гибкость реализации. Была изучена подробная документация по каждому алгоритму. Проверялась ее полнота и понятность, которая необходима при реализации сторонними разработчиками.

После 1-го этапа, в результате общего анализа, были отсеяны следующие алгоритмы: Frogbit, Fubuki, MAG, Mir-1, SSS, TRBDK3 YAEA.

2-й этап конкурса (2 августа 2006 – апрель 2007) включал в себя анализ алгоритмов производительности на различных платформах, операционных системах и архитектурах процессоров. В результате подробного анализа были выявлены слабые алгоритмы, которые выбыли из проекта: Phelix, Py, ABC, Achterbahn, DICING, Hermes8, NLS, Polar Bear, Pomaranch, SFINKS, TSC-3, VEST, WG, Yamb, ZKCrypt.

3-й этап проекта (апреля 2007 – май 2008) был ознаменован выпуском книги «New stream cipher designs» (Новые конструкции поточного шифра), в которую вошли история разработки и описание всех криптографических алгоритмов, вышедших в финальную часть.

Для выбора алгоритмов-победителей была сформирована комиссия, в которую вошли разработчики и исследователи крупнейших компаний и университетов Европы: С. Бэббидж (Vodafone, Великобритания), К. Де Каннье, Б. Пренел и У Хунцзюнь (Левенский католический университет, Бельгия), А. Канто (INRIA, Франция), К. Сид (Лондонский университет, Великобритания), А. Жильбер и М. Робшоу (France Telecom R&D, Франция), Т. Юханссон (Лундский университет, Швеция), К. Пар (Рурский университет, Германия), М. Паркер (Бергенский университет, Норвегия), В. Реймен (Технический университет Граца, Австрия).

Комиссия изучила всю имеющуюся документацию по алгоритмам, выслушала презентации авторов, и 15 апреля 2008 года, на основании тщательного анализа, были объявлены алгоритмы-победители проекта eStream в обеих категориях.

В категории «Программно-ориентированные алгоритмы» победителями стали: HC128, Rabbit, Salsa20, Sosemanuk.

В категории «Аппаратно-ориентированные алгоритмы» победителями стали: Grain, MICKEY, Trivium, F-FCSR.

В сентябре 2008 года криптографический шифр F-FCSR был исключен из финальной публикации в виду криптографической слабости, обнаруженной после завершения проекта.

Следующие разделы будут посвящены алгоритмам шифрования, которые стали победителями проекта eStream.

## 1.3 Алгоритм шифрования Salsa20

Salsa20 – алгоритм поточного шифрования информации, разработанный Даниэлем Бернштейном. Алгоритм был впервые представлен на проекте eStream, где стал победителем в категории «Программно-ориентированные алгоритмы».

Шифр использует хэш-функцию с 20-ю циклами, в которых поочередно применяются операции побитового сложения и циклического сдвига 32-битных машинных слов, которая обеспечивает криптостойкость алгоритма.

Основные характеристики алгоритма:

1. длина ключа не должна превышать 256 бит. В зависимости от длины ключа (до 128 бит или 256 бит) алгоритм будет видоизменяться;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. в функции расширения ключа (в зависимости от длины ключа) используется константа: «expand 16-byte key» или «expand 32-byte key»;
5. на вход алгоритму подается уникальный 64-битный вектор инициализации.

Согласно официальной документации, в алгоритме Salsa20 определены несколько стандартных функций, определяющих ключевые преобразования: quarterround, rowround, columnround, doubleround, salsa20 hash function и salsa20 expansion function.

Базовые операции представляют собой следующие преобразования.

Функция quarterround осуществляет преобразование над 4-мя 32-битными словами, при этом для каждого слова складываются 2 предыдущих. Затем полученная сумма сдвигается на определенное количество бит и полученный результат побитого суммируется с исходным словом. Математически функция quarterround(y) выглядит следующим образом:

*z*1 = *y*1 ⊕ ((*y*0 + *y*3) <<< 7);

*z*2 = *y*2 ⊕ ((*z*1 + *y*0) <<< 9);

*z*3 = *y*3 ⊕ ((*z*2 + *z*1) <<< 13);

*z*0 = *y*0 ⊕ ((*z*3 + *z*2) <<< 18),

где *z*i и *y*i – это32-битные слова.

В функцию rowround передается 16 слов, которые представляются в виде матрицы 4х4. Затем каждый ряд этой матрицы передается в функцию quarterround(*y*). Слова из строки берутся по порядку, начиная с *i*-го для *i*-ой строки, где *i* = {0, 1, 2, 3}. Математически функция rowround(*y*) может быть представлена:

(*z*0, *z*1, *z*2, *z*3) = quarterround(*y*0, *y*1, *y*2, *y*3);

(*z*5, *z*6, *z*7, *z*4) = quarterround(*y*5, *y*6, *y*7, *y*4);

(*z*10, *z*11, *z*8, *z*9) = quarterround(*y*10, *y*11, *y*8, *y*9);

(*z*15, *z*12, *z*13, *z*14) = quarterround(*y*15, *y*12, *y*13, *y*14),

где *y* = (*y*0, …, *y*15) – 16-битные слова, *z* = (*z*1, …, *z*15) – 16-битные слова, получаемые на выходе функции rowround(*y*).

На вход функции columnround(*y*) подаются 16 32-битных слов, которые представляются в виде матрицы 4х4, аналогично функции rowround. Затем каждый столбец этой матрицы подается в функцию quarterround(*y*). Слова из столбца берутся по порядку, начиная с *j*-го для *j*-го столбца, где *j* = {0, 1, 2, 3}. С помощью формул функцию columnround(*y*) можно представить следующим образом:

(*y*0, *y*4, *y*8, *y*12) = quarterround(*x*0, *x*4, *x*8, *x*12);

(*y*5, *y*9, *y*13, *y*1) = quarterround(*x*5, *x*9, *x*13, *x*1);

(*y*10, *y*14, *y*2, *y*6) = quarterround(*x*10, *x*14, *x*2, *x*6);

(*y*15, *y*3, *y*7, *y*11) = quarterround(*x*15, *x*3, *x*7, *x*11),

где *x* = (*x*1, …, *x*15) – 16-битные слова, *y* = (*y*0, …, *y*15) – 16-битные слова, получаемые на выходе функции columnround(*y*).

Функция doubleround(*y*) является последовательным выполнением функций columnround(*y*) и rowround(*y*). В общем виде данное преобразование можно записать так:

doubleround(*x*) = rowround(columnround(*y*)),

где *x* и *y* – это последовательность из 16-ти 32-битных слов.

Хэш-функция предназначена для генерирования уникальной 64-битной ключевой последовательности.

На вход функции подается 64-байтная последовательность. Затем каждые 4 байта подвергаются операции приведения к обратному порядку байт (littleendian). Математически эта операция выглядит так:

littleendian(*b*) = *b*0 + 28·*b*1 + 216·*b*2 + 224·*b*3,

где *b* = (*b*0, *b*1, *b*2, *b*3) – 32-битовое слово, то есть упорядоченная последовательность из 4 байт (*b*0, *b*1, *b*2, *b*3).

На выходе преобразования littleendian получается последовательность, состоящая из 16-ти 32-битных слов, которая 20 раз подается на вход функции doubleround.

На выходе хэш-функции salsa20 получается 512-ти битовая ключевая последовательность, которая используется при шифровании/расшифровывании потоков данных.

На вход функции расширении ключа подается секретный ключ (до 256 бит) и 128-битный вектор инициализации.

В функции используются 2 константы: «expand 16-byte key» (если длина ключа до 128 бит) или «expand 32-byte key» (если длина ключа до 256 бит).

Расширение ключа происходит за счет последовательной записи секретного ключа, вектора инициализации и константы в 512-ти битовую последовательность, которая передается в хэш-функцию для генерации ключевой последовательности. Данная 512-ти битовая последовательность схематично выглядит следующим образом:

для ключа до 128 бит: key\_exp = (*σ*0, *k*, *σ*1, *n*, *σ*2, *k*, *σ*3);

для ключа до 256 бит: key\_exp = (*σ*0, *k*0, *σ*1, n, *σ*2, *k*1, *σ*3),

где *k* – секретный ключ, либо его *i*-часть, *σ*i – константа (определяется в зависимости от длины ключа), *n* – вектор инициализации.

## 1.4 Алгоритм шифрования Rabbit

Rabbit – алгоритм поточного шифрования информации, впервые был представлен широкой публике в феврале 2003 года. В мае 2005 года Rabbit был заявлен для участия в проекте eStream, где стал победителем в категории «Программно-ориентированные алгоритмы».

Разработчиками алгоритма являются М. Боесгард, М. Вестэрагер, Т. Кристиансен и Э. Зеннер.

Основные характеристики алгоритма:

1. длина секретного ключа до 128 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 64-битный вектор инициализации.

Согласно официальной документации в алгоритме Rabbit определены несколько стандартных функций, определяющих ключевые преобразования: key\_setup, iv\_setup, next\_state.

Рассмотрим базовые преобразования. Функция key\_setup предназначена для установления ключа в память компьютера по следующему алгоритму. Секретный ключ разбивается на 8 подключей *k*0, *k*1, …, *k*7, на основании которых генерируются 8 переменных состояний *x*j,0 и 8 счетчиков состояний *с*j,0, по следующей схеме:

*x*j,0 = *k*(j+1 mod 8) || *k*j, где *j* четно;

*x*j,0 = *k*(j+5 mod 8) || *k*(j+4 mod 8), где *j* нечетно;

*c*j,0 = *k*(j+4 mod 8) || *k*(j+5 mod 8), где *j* четно;

*c*j,0 = *k*j || *k*(j+1 mod 8), где *j* нечетно,

где операция || означает конкатенацию 2-х подключей, *j* = {0, 1, …, 7}.

Вся схема проходит подобные преобразования 4 раза с помощью функции следующего шага (next\_state).

После чего все счетчики состояний переинициализируются согласно следующей формуле:

*c*j,4 = *c*j,4 ⊕ *x*(j+4 mod 8),4

После функции key\_setup следует функция iv\_setup, которая предназначена для установления вектора инициализации в память компьютера по следующему алгоритму:

*c*0,4 = *c*0,4 ⊕ IV[31..0];

*c*2,4 = *c*2,4 ⊕ IV[63..32];

*c*4,4 = *c*4,4 ⊕ IV[31..0];

*c*6,4 = *c*6,4 ⊕ IV[63..32];

*c*1,4 = *c*1,4 ⊕ (IV[63..48] × IV[31..16]);

*c*3,4 = *c*3,4 ⊕ (IV[47..32] × IV[15..0]);

*c*5,4 = *c*5,4 ⊕ (IV[63..48] × IV[31..16]);

*c*7,4 = *c*7,4 ⊕ (IV[47..32] × IV[15..0]),

где IV[31..0] – это последовательность бит (с 0 по 31) вектора инициализации.

Вся схема проходит подобные преобразования 4 раза с помощью функции следующего шага, которая так же лежит в основе генерации ключевой последовательности.

Функция next\_state содержит следующие уравнения:

*x*0,i+1 = *g*0,i + (*g*7,i <<< 16) + (*g*6,i <<< 16);

*x*1,i+1 = *g*1,i + (*g*7,i <<< 16) + *g*7,i;

*x*2,i+1 = *g*2,i + (*g*1,i <<< 16) + (*g*0,i <<< 16);

*x*3,i+1 = g3,i + (*g*7,i <<< 16) + *g*1,i;

*x*4,i+1 = *g*4,i + (*g*3,i <<< 16) + (*g*2,i <<< 16);

*x*5,i+1 = *g*5,i + (*g*7,i <<< 16) + *g*3,i;

*x*6,i+1 = *g*6,i + (*g*5,i <<< 16) + (*g*4,i <<< 16);

*x*7,i+1 = *g*7,i + (*g*7,i <<< 16) + *g*5,i,

где *x*i,j – переменные состояния, функция *g*j,i = ((*x*i,j + *c*j,i+1)2 ⊕ ((*x*i,j + *c*j,i+1)2 >> 32)) mod 232.

Система счетчиков так же подвержена изменением согласно функции следующего шага:

*c*0,i+1 = *c*0,i+1 + *a*0 + *φ*7,i mod 232;

*c*1,i+1 = *c*1,i+1 + *a*1 + *φ*0,i+1 mod 232;

*c*2,i+1 = *c*2,i+1 + *a*2 + *φ*1,i+1 mod 232;

*c*3,i+1 = *c*3,i+1 + *a*3 + *φ*2,i+1 mod 232;

*c*4,i+1 = *c*4,i+1 + *a*4 + *φ*3,i+1 mod 232;

*c*5,i+1 = *c*5,i+1 + *a*5 + *φ*4,i+1 mod 232;

*c*6,i+1 = *c*6,i+1 + *a*6 + *φ*5,i+1 mod 232;

*c*7,i+1 = *c*7,i+1 + *a*7 + *φ*6,i+1 mod 232,

где функция *φ*j,i+1 определяется следующим образом:

*φ*j,i+1 = 1, если *c*0,i + *a*0 + *φ*7,i  ≥ 232;

*φ*j,i+1 = 1, если *c*j,i + *a*j + *φ*j-1,i+1 ≥ 232;

*φ*j,i+1 = 0, если другое.

Константы *a*i определены согласно официальной документации:

*a*0 = 0x4D34D34D; *a*1 = 0xD34D34D3;

*a*2 = 0x34D34D34; *a*3 = 0x4D34D34D;

*a*4 = 0xD34D34D3; *a*5 = 0x34D34D34;

*a*6 = 0x4D34D34D; *a*7 = 0xD34D34D3.

В результате работы функций next\_state генерируется уникальная ключевая последовательность по следующей схеме:

*s*i[15..0] = *x*0,i[15..0] ⊕ *x*5,i[31..16]; *s*i[31..16] = *x*0,i[31..16] ⊕ *x*3,i[15..0];

*s*i[47..32] = *x*2,i[15..0] ⊕ *x*7,i[31..16]; *s*i[63..48] = *x*2,i[31..16] ⊕ *x*5,i[15..0];

*s*i[15..0] = *x*4,i[15..0] ⊕ *x*1,i[31..16]; *s*i[95..80] = *x*4,i[31..16] ⊕ *x*7,i[15..0];

*s*i[15..0] = *x*6,i[15..0] ⊕ *x*3,i[31..16]; *s*i[127..112] = *x*6,i[31..16] ⊕ *x*1,i[15..0],

где *s*i – 16 бит ключевой последовательности.

Ключевая последовательность используется для операции побитового сложения по модулю 2 с каждым байтом входного потока. Вследствие чего на выходе получается зашифрованный поток данных.

Функция расшифровывания аналогична функции шифрования.

## 1.5 Алгоритм шифрования HC-128

HC – алгоритм поточного шифрования информации, разработанный криптографом У Хунцзюн. 128-битный вариант реализации алгоритма был заявлен для участия в проекте eStream, где стал победителем в категории «Программно-ориентированные алгоритмы».

Основные характеристики алгоритма:

1. длина ключа не должна превышать 128 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 128-битный вектор инициализации.

Согласно официальной документации в алгоритме HC-128 определены несколько стандартных функций, определяющих базовые преобразования: функция инициализации и функция генерирования ключевой последовательности.

В алгоритме HC-128 используется 6 стандартных преобразований: *f*1, *f*2, *g*1, *g*2, *h*1,*h*2. На математическом языке данные преобразования можно представить так:

*f*1(*x*) = (*x* >>> 7) ⊕ (*x* >>> 18) ⊕ (*x* >> 3);

*f*2(*x*) = (*x* >>> 17) ⊕ (*x* >>> 19) ⊕ (*x* >> 10);

*g*1(*x*, *y*, *z*) = ((*x* >>> 10) ⊕ (*z* >>> 23)) + (*y* >>> 8);

*g*2(*x*, *y*, *z*) = ((*x* <<< 10) ⊕ (*z* <<< 23)) + (*y* <<< 8);

*h*1(*x*) = *Q*[*x*0] + *Q*[256+*x*2];

*h*2(*x*) = *P*[*x*0] + *P*[256+*x*2],

где *x*, *y*, *z* – 32-битные машинные слова, *P* и *Q* – массивы 32-битных слов.

Функция инициализации предназначена для записи и расширения секретного ключа и вектора инициализации в массивы *P* и *Q* соответственно. Схематично данный процесс можно представить так:

*W*i = Ki, при 0 ≤ *i* ≤ 7;

*W*i = IVi, при 8 ≤ *i* ≤ 15;

*W*i = *f*2(*W*i-2) + *W*i-7 + *f*1(*W*i-15) + *W*i-16 + *i*, при 16 ≤ *i* ≤ 1279;

*P*[*i*] = *W*i+256, *Q*[*i*] = *W*i+768, при 0 ≤ *i* ≤ 511;

Система начинает процесс шифрования 1024 раза без генерации ключевой последовательности:

*P*[*i*] = (*P*[*i*] + *g*1(*P*[*i*-3], *P*[*i*-10], *P*[*i*-511]) ⊕ *h*1(*P*[*i*-12])), при 0 ≤ *i* ≤ 511;

*Q*[*i*] = (*Q*[*i*] + *g*1(*Q*[*i*-3], *Q*[*i*-10], *Q*[*i*-511]) ⊕ *h*1(*Q*[*i*-12])) при 0 ≤ *i* ≤ 511,

где операция «-» – операция вычитания по модулю 512.

Функцию генерирования ключевой последовательности можно представить так:

при (*i* mod 1024) < 512

*P*[*i*] = (*P*[*i*] + *g*1(*P*[*i*-3], *P*[*i*-10], *P*[*i*-511]);

*s* = *h*1(*P*[*i*-12]) ⊕ *P*[*i*].

при (*i* mod 1024) ≥ 512

*Q*[*i*] = (*Q*[*i*] + *g*1(*Q*[*i*-3], *Q*[*i*-10], *Q*[*i*-511]);

*s* = *h*2(*Q*[*i*-12]) ⊕ *Q*[*i*],

где *s*i – 32-битный элемент ключевой последовательности, *i* – длина входного потока данных.

## 1.6 Алгоритм шифрования Sosemanuk

Sosemanuk – алгоритм поточного шифрования информации. Был заявлен для участия в проекте eStream, где стал победителем в категории «Программно-ориентированные алгоритмы».

Разработчиками алгоритма являются К. Бербайн, О. Биллет, А. Кантейт, Н. Кортойс, Х. Гилберт, Л. Губин, А. Гойджет, Л. Гранболан, К. Люрадойкс, М. Миниер, Т. Порнин и Х. Сиберт.

Основные характеристики алгоритма:

1. длина ключа не должна превышать 256 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 128-битный вектор инициализации.

Согласно официальной документации в алгоритме Sosemanuk определены несколько стандартных функций, определяющих ключевые преобразования: функции расширения ключа и вектора инициализации, функция генерирования ключевой последовательности.

Рассмотрим основные базовые преобразования. Функция расширения ключа основана на разбиении секретного ключа на 8 подключей по 32 бита каждый. Затем все подключи побитового складываются по модулю 2 и полученный результат циклически сдвигается на 11 бит. Затем массив подключей подается на вход 1 раунду блочного алгоритма Serpent. Алгоритм расширения ключа насчитывает 25 подобных преобразований, на выходе которых генерируется 100 32-битных слов. Схема расширения ключа приведена ниже:

*w*0 = K0, …, K31;

*w*1 = K32, …, K63;

.

.

.

*w*7 = K224, …, K255;

*tt* = *w*i ⊕ *w*i+3 ⊕ *w*i+5 ⊕ *w*i+7 при *i* ∍ {0, …, 7};

*w*i = *tt* <<< 11;

serpent(*w*),

где *w*i – *i*-й 32-х битный подключ, Ki – *i*-й бит ключа, serpent – расширение ключа блочного алгоритма Serpent (**тут ссылка будет**), *tt* – переменная для промежуточных вычислений.

Функция расширения вектора инициализации основана на использовании блочного шифра Serpent. Вектор инициализации разбивается на 4 32-х битных вектора, которые подаются на вход 1-му раунду шифра Serpent. Вся схема проходит 24 итерации, что позволяет достичь криптостойкости алгоритма. Расширение вектора инициализации можно представить так:

*r*0 = IV0, …, IV31;

*r*1 = IV32, …, IV63;

*r*2 = IV64, …, IV95;

*r*3 = IV96, …, IV127;

serpent24(*r*),

где *r*i – *i*-й 32-х битный вектор, IVi – *i*-й бит вектора инициализации, serpent24 – 24 раунда блочного алгоритма Serpent.

Функция генерации ключевой последовательности осуществляет преобразования над массивами байт полученных при работе функций расширения ключа и вектора инициализации. Функцию генерации ключевой последовательности на псевдокоде можно записать следующим образом:

for *i* = 0 to 4

*tt* = XMUX(*r1*, *s*j, *s*t);

*or1* = *r1*;

*r1* = *r2* + *tt*;

*tt* = 0x54655307 · *r1*;

*r2* = *tt* <<< 7;

*v*i = *s*t;

*u*i = (*s*j + r1) ⊕ *r2*;

end for,

*z*0 = (*u*2 ⊕ *v*0);

*z*1 = (*u*3 ⊕ *v*1);

*z*2 = (*u*1 ⊕ *v*2);

*z*3 = (*u*4 ⊕ *v*3),

где *s* – массив битов, полученный путем расширения секретного ключа и вектора инициализации, *z* – поток ключевой последовательности, *u*i, *v*i, *r1*, *r2*, *tt*, *or1* – переменные для промежуточных вычислений, *i* и *j* – индексы, генерируемые в зависимости от раунда алгоритма, функция XMUX(*r1*, *s*j, *s*t) возвращает *s*j, если *r1*= 0, или *s*t ⊕*s*j, если *r1* = 1.

## 1.7 Алгоритм шифрования Grain

Grain – алгоритм поточного шифрования информации, разработанный Т. Юханссоном, А. Максимовым и М. Хеллом. Версия со 128-битным ключом была заявлена для участия в проекте eStream, где стала победителем в категории «Аппаратно-ориентированные алгоритмы».

Основные характеристики алгоритма:

1. длина ключа не должна превышать 128 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 96-битный вектор инициализации.

Согласно официальной документации в алгоритме Grain определены несколько стандартных функций, определяющих базовые преобразования: регистр сдвига с линейной обратной связью, регистр сдвига с нелинейной обратной связью и выходная функция.

Базовые операции представляют собой следующие преобразования.

Секретный ключ разбивается на биты и загружается в регистр сдвига с нелинейной обратной связью (РССНОС), а вектор инициализации – в регистр сдвига с линейной обратной связью (РССЛОС).

РССЛОС, объемом 96 бит, предназначен для генерации нового бита на основе битов вектора инициализации. Формула для генерирования нового бита:

*s*i+96 = *s*0 ⊕ *s*7 ⊕ *s*38 ⊕ *s*70 ⊕ *s*81 ⊕ *s*96,

где *s*i – *i*-ый бит РССЛОС.

РССНОС, объемом 128 бит, предназначен для генерирования нового бита на основе битов секретного ключа и 1-го бита регистра с линейной обратной связью. Формула для генерации нового бита:

*b*i+128 = *s*i ⊕ *b*i + *b*26 ⊕ *b*56 ⊕ *b*91⊕ *b*96 ⊕ (*b*3 & *b*67)⊕ (*b*11 & *b*13) ⊕ (*b*17 & *b*18) ⊕ (*b*27 & *b*59) ⊕ (*b*40 & *b*48) ⊕ (*b*61 & *b*65) ⊕ (*b*68 & *b*84),

где *b*i – *i*-ый бит регистра РССНОС, *s*i – *i*-ый бит РССЛОС.

При старте работы алгоритма, генерация новых битов с помощью РССНОС и РССЛОС, повторяется 256 раз без генерации битов ключевого потока. Весь регистр сдвигается на 1-ну позицию влево, а новые сгенерированные биты записываются в последние позиции соответствующих регистров.

Выходом выходной функции является бит ключевой последовательности, который вычисляется по следующей формуле:

*z*i = j+i ⊕ *h*(*x*) ⊕ *s*i+93 при *A* = {2, 15, 36, 45, 64, 73, 89},

где *b*i – *i*-ый бит регистра РССНОС, *s*i – *i*-ый бит РССЛОС, *h*(*x*) = (*b*12 & *s*8) ⊕ (*s*13 & *s*20) ⊕ (*b*95 & *s*42) ⊕ (*s*60 & *s*79) ⊕ (*b*12 & *b*95 & *s*95),

Таким образом, за одну итерацию алгоритма генерируется 1 бит ключевой последовательности.

## 1.8 Алгоритм шифрования Trivium

Trivium – алгоритм поточного шифрования информации, разработанный К. де Канньером и Б. Пренелом. Шифр был заявлен для участия в проекте eStream, где стал победителем в категории «Аппаратно-ориентированные алгоритмы».

Основные характеристики алгоритма:

1. длина ключа не должна превышать 80 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 80-битный вектор инициализации.

Согласно официальной документации в алгоритме Trivium определены несколько стандартных функций, определяющих базовые преобразования: функция инициализации и функция генерации ключевой последовательности.

Базовые операции представляют собой следующие преобразования.

Функция инициализации осуществляет преобразование 80-ти битного секретного ключа и вектора инициализации в определенную последовательность бит и затем их трансформацию для дальнейшей генерации ключевого потока.

Схематично функцию инициализации на псевдокоде можно представить так:

(*s*1, *s*2, …, *s*93) = (K1, …, K80, 0, …, 0);

(*s*94, *s*95, …, *s*177) = (IV1, …, IV80, 0, …, 0);

(*s*178, *s*179, …, *s*288) = (0, …, 0, 1, 1, 1);

for *i* = 1 to 4 \* 288 do

*t*1 = *s*66 ⊕ *s*91 & *s*92 ⊕ *s*93 ⊕ *s*171;

*t*2 = *s*161 ⊕ *s*175 & *s*176 ⊕ *s*177 ⊕ *s*264;

*t*3 = *s*243 ⊕ *s*286 & *s*287 ⊕ *s*288 ⊕ *s*69;

(*s*1, *s*2, …, *s*93) = (*t*3, *s*1, *s*2, …, *s*92);

(*s*94, *s*95, …, *s*177) = (*t*1, *s*94, *s*95, …., *s*176);

(*s*178, *s*179, …, *s*288) = (*t*2, *s*178, *s*179, …, *s*287);

end for,

где *s* – 288-ми битовый регистр, Ki – *i*-й бит ключа, IVi – *i*-й бит вектора инициализации, *t*1, *t*2, *t*3 – биты для промежуточных вычислений.

После вышеописанного преобразования регистр *s* подается на вход функции генерации ключевой последовательности, алгоритм которой на псевдокоде приведен ниже:

for *i* = 1 to *N* do

*t*1 = *s*66 ⊕ *s*93;

*t*2 = *s*162 ⊕ *s*177;

*t*3 = *s*243 ⊕ *s*288;

*z*i = *t*1 ⊕ *t*2 ⊕ *t*3;

*t*1 = *t*1 ⊕ *s*91 & *s*92 ⊕ *s*171;

*t*2 = *t*2 ⊕ *s*175 & *s*176 ⊕ *s*264;

*t*3 = *t*3 ⊕ *s*286 & *s*287 ⊕ *s*69;

(*s*1, *s*2, …, *s*93) = (*t*3, *s*1, *s*2, …, *s*92);

(*s*94, *s*95, …, *s*177) = (*t*1, *s*94, *s*95, …., *s*176);

(*s*178, *s*179, …, *s*288) = (*t*2, *s*178, *s*179, …, *s*287);

end for,

где *N* – длина входного потока в битах, *z*i – бит ключевой последовательности.

## 1.9 Алгоритм шифрования Mickey

Mickey – алгоритм поточного шифрования информации, разработанный С. Бэббиджем и М. Доддомю Шифр был заявлен для участия в проекте eStream, где стал победителем в категории «Аппаратно-ориентированные алгоритмы».

Основные характеристики алгоритма:

1. длина ключа не должна превышать 80 бит;
2. входной поток данных не ограничен по объему;
3. выходной поток данных такого же объема, как и входной;
4. на вход алгоритму подается уникальный 80-битный вектор инициализации.

Согласно официальной документации в алгоритме MICKEY определены несколько стандартных функций, определяющих базовые преобразования: сдвиг регистра R (CLOCK\_R), сдвиг регистра S (CLOCK\_S), функция управления алгоритмом (CLOCK\_KG), функция загрузки ключа и вектора инициализации и функция генерирования ключевой последовательности.

Рассмотрим базовые преобразования. Функция сдвига регистра *R* принимает несколько параметров: CLOCK\_R (*R*, *input\_bit\_r*, *control\_bit\_r*). Здесь *R* – это указатель на сам регистр, а *input\_bit\_r* и *control\_bit\_r* – специальные биты. Данное преобразование на псевдокоде выглядит следующим образом:

CLOCK\_R (*R*, *input\_bit\_r*, *control\_bit\_r*)

*feedback\_bit* = *r*99 ⊕ *input\_bit\_r*;

для 1 ≤ *i* ≤ 99, *r*'i = *r*i-1; *r*'0 = 0;

для 0 ≤ *i* ≤ 99, если *i* ∈ *RTAPS*, *r*'i = *r*'i ⊕ *feedback\_bit*;

если *control\_bit\_r* = 1, то

для 0 ≤ *i* ≤ 99, *r*'i = *r*'i ⊕ *r*i,

где *r*0, *r*1, …, *r*99 – биты регистра *R* до сдвига, *r*'1, *r*'2, …, *r*'99 – биты регистра *R* после сдвига, *RTAPS* – набор входов обратной связи.

Массив *RTAPS* определен следующим образом:

*RTAPS* = { 0, 1, 3, 4, 5, 6, 9, 12, 13, 16, 19, 20, 21, 22, 25, 28, 37, 38, 41, 42, 45, 46, 50, 52, 54, 56,58, 60, 61, 63, 64, 65, 66, 67, 71, 72, 79, 80, 81, 82, 87, 88, 89, 90, 91, 92, 94, 95, 96, 9}.

Функция сдвига регистра S так же принимает несколько параметров: CLOCK\_S(*S*, *input\_bit\_s*, *control\_bit\_s*). Здесь *S* – это указатель на регистр, а *input\_bit\_s*, *control\_bit\_s* – специальные биты. Сдвиг регистра *S* на псевдокоде выглядит следующим образом:

CLOCK\_S(*S*, *input\_bit\_s*, *control\_bit\_s*)

*feedback\_bit* = *s*99 ⊕ *input\_bit\_s*;

для 1 ≤ *i* ≤ 98,

*ŝ*i = *s*i-1 ⊕ ((*s*i ⊕ *COMP0*i)&(*s*i+1 ⊕ *COMP1*i)); *ŝ*0 = 0; *ŝ*99 = *s*98;

если *control\_bit\_s* = 0, то

для 0 ≤ *i* ≤ 99, s'i = ŝi ⊕ (*FB0*i & *feedback\_bit*);

если *control\_bit\_s* = 1, то

для 0 ≤ *i* ≤ 99, *s'*i = *ŝ*i ⊕ (*FB1*i & *feedback\_bit*),

где *s*0, *s*1, …, *s*99 –биты регистра *S* до сдвига, *ŝ*0, *ŝ*1, …, *ŝ*99 – промежуточные значения битов регистра, *s'*0, *s'*1, …, *s'*99 – биты регистра *S* после сдвига.

Массивы *COMP0*, *COMP1*, *FB0* и *FB1*, в зависимости от раунда, определяются по таблицам смещения. Познакомиться с таблицами можно по следующей ссылке – <http://ww.ecrypt.eu.org/stream/mickey/>.

Порядок работы алгоритма регулируется функцией управления. Функция принимает 4 параметра: CLOCK\_KG(*R*, *S*, *mixing*, *input\_bit*). Здесь *R* и *S* указатели на регистры *R* и *S* соответственно, а *mixing* и *input\_bit* являются служебными битами. На псевдокоде данную функцию можно записать так:

CLOCK\_KG(*R*, *S*, *mixing*, *input\_bit*)

*control\_bit\_r* = *s*34 ⊕ *r*67;

*control\_bit\_s* = *s*67 ⊕ *r*33;

если *mixing* = TRUE, тогда *input\_bit\_r* = *input\_bit* ⊕ *s*50,

а если *mixing* = FALSE , тогда *input\_bit\_r* = *input\_bit*;

*input\_bit\_s* = *input\_bit*;

CLOCK\_R (*R*, *input\_bit\_r*, *control\_bit\_r*);

CLOCK\_S (*S*, *input\_bit\_s*, *control\_bit\_s*).

На основании вышеописанных преобразований при начале работы алгоритма происходит загрузка секретного ключа и вектора инициализации в регистры *R* и *S* соответственно. На псевдокоде данную функцию можно записать так:

(*r*0, *r*1, …, *r*99)= (0, …, 0);

(*s*0, *s*1, …, *s*99) = (0, …, 0);

для 0 ≤ *i* ≤ длина IV -1,

CLOCK\_KG (*R*, *S*, *mixing* = TRUE, *input\_bit* = IV i);

для 0 ≤ *i* ≤ 79,

CLOCK\_KG (*R*, *S*, *mixing* = TRUE, *input\_bit* = Ki);

для 0 ≤ *i* ≤ 99,

CLOCK\_KG (*R*, *S*, *mixing* =TRUE, *input\_bit* = 0),

где *K*i – *i*-й бит секретного ключа, IV i – *i*-й бит вектора инициализации.

Функция генерирования ключевой последовательности на псевдокоде выглядит следующим образом:

для 0 ≤ *i* ≤ *L* − 1,

*z*i = *r*0 ⊕ *s*0;

CLOCK\_KG (*R*, *S*, *mixing* = FALSE, *input\_bit* = 0),

где *L* – длина входного потока, *z*i – *i*-й бит ключевой последовательности.

## 1.10 Общая схема функции шифрования/расшифровывания

Алгоритмы проекта eStream за одну итерацию генерируют последовательность байт, называемую ключевой последовательностью. Для каждого шифра длина ключевой последовательности определена согласно официальной документации. Эта последовательность байт используется для побитового сложения по модулю 2 с каждым байтом входного потока данных. Вследствие чего на выходе получается шифрованный поток данных.

Функция расшифровывания аналогична функции шифрования. Процесс шифрования/расшифровывания изображен на рисунке 1.

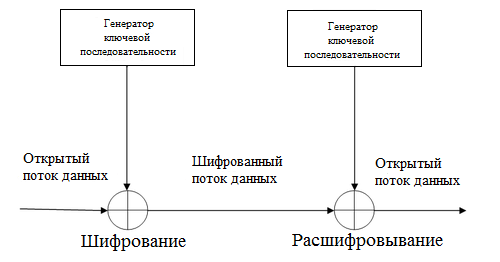


Рисунок 1 – Процесс шифрования/расшифровывания

## 1.11 Алгоритмы проекта eStream в действующих проектах

В виду большой скорости шифрования входного потока и повышенной криптостойкости, которая была заложена в алгоритмы, область применения шифров проекта eStream достаточно разнообразна:

* 1. шифрование потоков данных в режиме реального времени (системы мгновенного обмена сообщениями, видеоконференции, защищенный канал передачи данных и другое);
  2. шифрование информации в режиме блочных шифров (шифрование файлов, разделов жестких дисков и другое);
  3. аппаратные комплексы шифрования (организация защищенных каналов связи);
  4. использование алгоритмов как составной части системы шифрования (как промежуточная стадия шифрования информации или как одновременное использование нескольких алгоритмов шифрования).

Такая обширная область применения потоковой криптографии не могла не заинтересовать разработчиков программного обеспечения.

Результаты работы проекта eStream используются в криптографической библиотеке «Crypto++», распространяющейся под лицензией «Boost Software License 1.0». Проект «Crypto++» является кроссплатформенным, который реализован преимущественно на языке программирования C++.

Проект объединяет в себе несколько типов криптографических алгоритмов: потоковые, блочные, хэш-функции, ассиметричные. Раздел потоковые шифры содержит алгоритмы проекта eStream из категории «программно-ориентированные алгоритмы»: Salsa и Sosemanuk.

Шифр HC128 используется в кроссплатформенной криптографической библиотеке «Bounce Castle», которая разрабатывается австралийскими исследователями. Библиотека разработана на языках программирования java и C# и распространяется под лицензией «Massachusetts Institute of Technology License». Последняя публикация проекта датируется мартом 2015 года.

Алгоритм Rabbit входит в состав веб-приложения для шифрования информации «Crypto Tools», которое распространяется под лицензией «GNU Ceneral Public License». Разработчиком является сообщество IBlogBox. Приложение встраивается в браузер в виде дополнительного расширения и предназначено преимущественно для шифрования небольших объемов информации. (<http://iblogbox.com/devtools/crypto/>)

Шифры Grain и Trivium используются в библиотеке «AVR Crypto lib», которая ориентирована преимущественно для работы на микроконтроллерах. Разработкой библиотеки занимается немецкая лаборатория «Das Labor». Библиотека реализована на языке программирования C и распространяется под лицензией «GPLv3».(<http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>)

Исходные коды алгоритмов проекта eStream, которые включены в состав вышеописанных проектов, используются в том виде, в котором их предложили авторы шифров, без добавления каких-либо модификаций.

В результате чего возникла идея о разработке единого проекта, который объединил в себе все алгоритмы-победители проекта eStream, предоставил разработчикам программного обеспечения удобный интерфейс взаимодействия с шифрами и представил новые модернизированные реализации алгоритмов с исправленными ошибками и недочетами.

## 1.12 Безопасность алгоритмов проекта eStream

В проекте eStream, наряду со скоростью шифрования информации, изучался аспект стойкости алгоритмов к современным методам атак. Для этих целей была разработана платформа для анализа шифров, так называемый «Testing framework». С описанием платформы можно ознакомиться по ссылке – <http://www.ecrypt.eu.org/stream/perf/> .

За время работы проекта криптографические шифры подверглись различным испытаниями и тестам: анализ свойств генератора ключевой последовательности; анализ процедур инициализации (процедуры установки секретного ключа и вектора инициализации); атака по времени; атака по сторонним каналам; дифференциальная атака и многие другие.

Все 7 алгоритмов-победителей успешно прошли испытания. Подробные результаты тестов, а так же отчеты и протоколы конференций, содержатся в книге «New Stream Cipher Designs» и на официальном сайте проекта eStream в разделе «Результаты тестирования» – <http://www.ecrypt.eu.org/stream/perf/#results>.

# ГЛАВА 2. БИБЛИОТЕКА estream.h

Последняя публикация алгоритмов-победителей проекта eStream датируется январем 2012 года.

Организаторы представили результаты конкурса как разрозненные проекты, независящие друг от друга. Единственным исключением является то, что функции, использующиеся в каждом шифре, называются одинаково. Но организаторами не было представлено единого механизма, который бы позволил использовать конкурсные разработки в рамках единого проекта. Сторонним разработчикам предлагается самим выбрать нужный алгоритм и интегрировать его с уже более большими проектами.

Возникла идея не только реализовать все 7 криптографических шифров, но и предоставить сторонним разработчикам программного обеспечения удобный интерфейс для облегчения интеграции алгоритмов шифрования в свои проекты.

## 2.1 Структура библиотеки estream.h

Библиотека получила название estream.h в честь проекта, который подарил миру новые криптографические алгоритмы. Она была разработана на языке программирования C.

Библиотека насчитывает 16 файлов. 9 заголовочных файлов: estream.h, macro.h, salsa.h, rabbit.h, hc128.h, sosemanuk.h, grain.h, mickey.h, trivium.h. И 7 исходных файлов: salsa.c, rabbit.c, hc128.c, sosemanuk.c, grain.c, mickey.c, trivium.c.

Структура библиотеки estream.h представлена на рисунке 2.

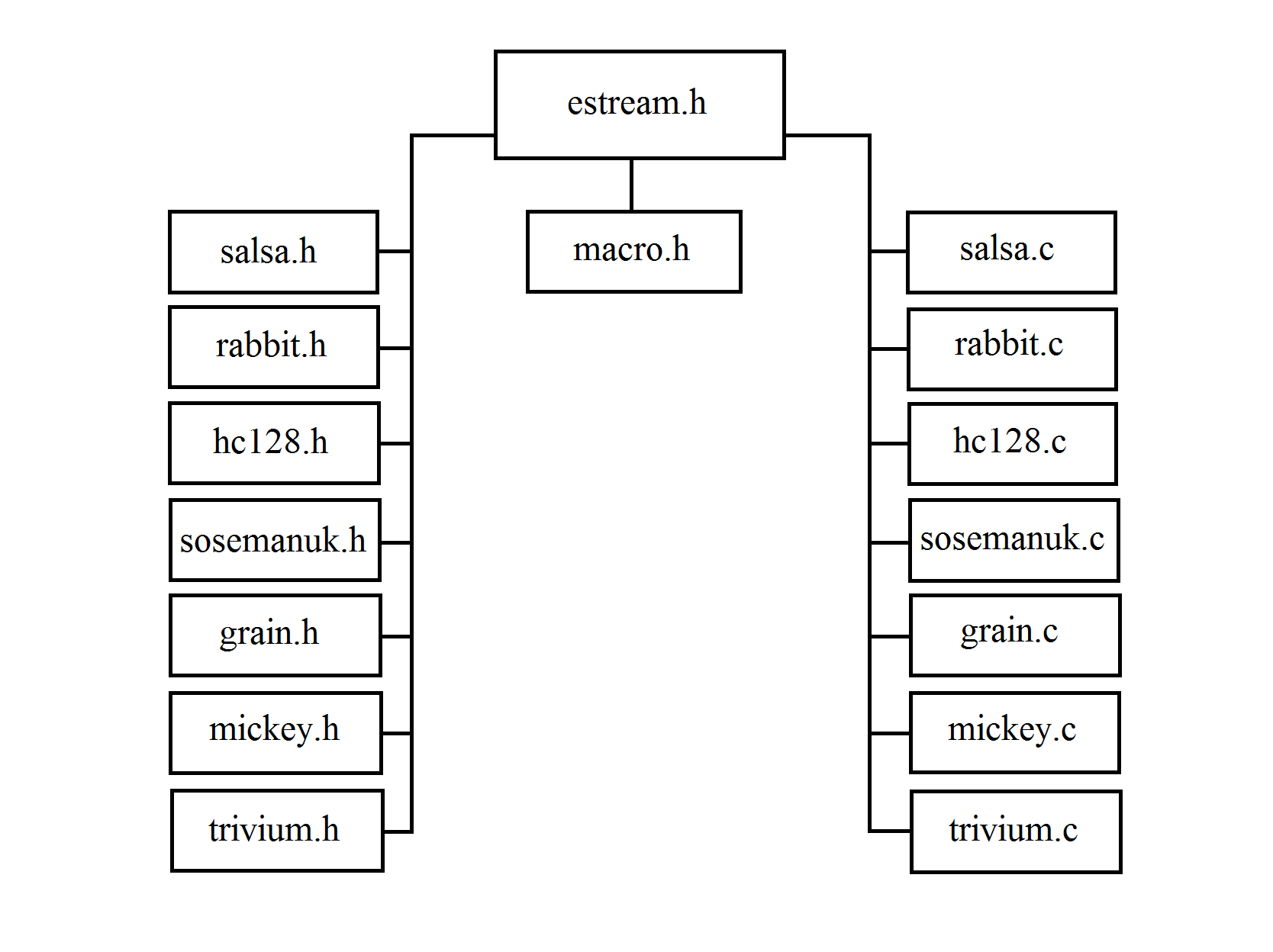


Рисунок 2 – Структура библиотеки estream.h

Основным файлом всего проекта является заголовочный файл estream.h, который необходимо разработчикам подключать к своей работе. Файл estream.h содержит инструкции, которые подгружают остальные заголовочные файлы. Исходный код заголовочного файла estream.h приведен в листинге 1.

Листинг 1 – Исходный код заголовочного файла estream.h

#ifndef ESTREAM\_H

#define ESTREAM\_H

#include «salsa.h»

#include «rabbit.h»

#include «hc128.h»

#include «sosemanuk.h»

#include «grain.h»

#include «mickey.h»

#include «trivium.h»

#endif

Подгружаемые в estream.h файлы содержат базовый набор функций доступных разработчикам и специально сформированные структуры данных. Все подгружаемые заголовочные файлы называются в зависимости от алгоритма шифрования, который они описывают. Например, файл salsa.h содержит исходный код, описывающий криптографический шифр Salsa.

Каждый заголовочный файл (header-файл) содержит описание структуры данных. Структура данных представляет собой объединение нескольких переменных. В листинге 2 приведен исходный код структуры данных для алгоритма Salsa (header-файл salsa.h).

Листинг 2 – Исходный код структуры данных salsa\_context

struct salsa\_context {

int keylen;

int ivlen;

uint8\_t key[32];

uint8\_t iv[16]

uint32\_t x[16];

}

Для каждого алгоритма в соответствующем header-файле заведена своя структура данных. Причем количество полей в структуре данных в каждом алгоритме разное, а общими являются следующие поля:

1. keylen – длина секретного ключа в байтах;

2. ivlen – длина вектора инициализации в байтах;

3. key – массив, содержащий биты секретного ключа;

4. iv – массив, содержащий биты вектора инициализации.

Использование структур данных позволяет облегчать взаимодействие разработчика с библиотекой. Так как один раз, заполнив структуру нужными значениями, разработчику нет необходимости заботиться о том, что происходит с этими значениями в процессе работы алгоритма.

Помимо структур данных, каждый header-файл содержит ссылки на 3 функции: \*\_set\_key\_and\_iv, \*\_crypt, \*\_test\_vectors (\* – обозначает один из 7-ми алгоритмов шифрования).

Функция \*\_set\_key\_and\_iv(struct \*ctx, const uint8\_t \*key, const int keylen, const uint8\_t \*iv, const int ivlen) предназначена для заполнения структуры данных битами секретного ключа и вектора инициализации. Перед заполнением структуры происходит ее инициализация (заполнение всех полей нулями). Это необходимо для устранения попадания «мусора» в поля структуры.

Функция принимает 5 аргументов: указатель на структуру данных, указатель на секретный ключ, длину секретного ключа, указатель на вектор инициализации и длину вектора инициализации. При успешном выполнении функция возвращает 0, в случае ошибки возвращает -1.

Функция \*\_crypt(struct \*ctx, const uint8\_t \*buf, uint32\_t buflen, uint8\_t \*out) шифрует/расшифровывает входной поток данных. Функция принимает на вход 4 аргумента: указатель на структуру данных, указатель на входной поток данных, длину входного потока в байтах и указатель на выходной массив данных, в который будет помещен результат шифрования/расшифровывания. Данная функция ничего не возвращает.

Функция \*\_test\_vectors(struct \*ctx) предназначена для вывода ключевого потока, который генерируется за одну итерацию алгоритма. Длина ключевого потока зависит от алгоритма. На вход функции подается указатель на уже заполненную ключом и вектором инициализации структуру данных. Функция ничего не возвращает.

Заголовочный файл macro.h содержит описания некоторых математических преобразований, которые являются общими для нескольких криптографических алгоритмов. Подобный подход позволил уменьшить дублирование исходного кода. В header-файле macro.h содержатся макросы для обеспечения кроссплатформенности библиотеки и макросы циклических сдвигов машинных слов.

Библиотека estream.h поддерживает два основных порядка байтов: big-endian и little-endian.

Big-endian (порядок от старшего к младшему) – запись машинного слова начинается со старшего разряда и заканчивается младшим. Подобный порядок байтов поддерживается процессорами IBM 360/370/390, MIPS, SPARC и другими моделями процессоров.

Little-endian (порядок от младшего к старшему) – запись машинного слова начинается с младшего разряда и заканчивается старшим. Подобный порядок байтов впервые стала использовать компания Intel, разработавшая процессоры с х86 архитектурой.

Если же в процессоре используется иной порядок байтов, что маловероятно, то библиотека выдаст ошибку «unsupported byte order» (порядок байт не поддерживается).

Наряду с заголовочными файлами библиотека содержит 7 файлов описания, которые содержат исходный программный код криптографических алгоритмов на языке программирования С. Файл описания носит такое же имя, как и заголовочный файл только с расширением .c. Например, пара: header-файл salsa.h и его файл описание salsa.c.

Разработанная библиотека estream.h предоставляет удобный интерфейс взаимодействия между разработчиком и криптографическими алгоритмами.

## 2.2 Механизмы модернизации

Главным достоинством потоковых криптографических алгоритмов является высокая скорость шифрования информации. Поэтому недостаточно просто реализовать шифр. Необходимо предложить такую реализацию, которая бы обеспечила высокую производительность алгоритма и смогла бы работать на различных архитектурах и платформах. Подобный подход был использован при разработке библиотеки.

Данный раздел посвящен механизмам и способам модернизации алгоритмов для увеличения скоростных показателей, достижения кроссплатформенности при стабильной и корректной работе шифров.

### 2.2.1 Использование макросов

Макрос (в терминологии языка программирования С) – символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций.

Макросы используются для уменьшения дублирования программного кода, либо для увеличения понимания исходного кода, когда макросом заменяется какая-либо часть сложной составной инструкции.

Когда препроцессор, при обработке файла с исходным кодом, встречает символьное имя, которое определено с помощью директивы #define, то он подставляет значение макроса вместо символьного имени.

Подобное поведение препроцессора можно использовать для увеличения скорости работы алгоритма. Вместо многократного использования функций, где время тратится на вызов функции, можно использовать макрос. При обработке файла препроцессор сам подставит необходимый программный код. Такой подход позволяет получить выигрыш по времени, если набирается большое количество вызовов функции.

Например, в алгоритме HC128 (hc128.c) используется макрос GENERATE\_P, его исходный код приведен в листинге 3. Данный макрос производит генерацию ключевого потока.

Листинг 3 – Исходный код макроса GENERATE\_P

#define GENERATE\_P(ctx, a, b, c, d, e, f, res) {

uint32\_T res1, res2;

G1(ctx->x[e], ctx->x[d], ctx->w[b], res1);

H1(ctx, ctx->x[f], res2);

ctx->w[a] += res1;

ctx->x[c] = ctx->w[a];

res = U32TO32((res2 ^ ctx->w[a]));

При шифровании 512 байт информации данный макрос вызывается 32 раза. Использование макроса в данном случае дает значительный выигрыш в скорости, по сравнению с вызовом аналогичной функции 32 раза. С ростом объемов данных выигрыш в скорости становится более ощутимым, что показывают проведенные исследования, представленные далее.

### 2.2.2 Использование битовых операций

Язык программирования С представляет удобный инструмент для работы с битами. В рамках разработки библиотеки использовались следующие битовые операции: левый битовый сдвиг, правый битовый сдвиг, побитовое отрицание, побитовое сложение и побитовое умножение.

Использование битовых операций позволяет увеличить производительность работы алгоритмов, так как все операции производятся непосредственно с битами машинного слова. Пример комплексного использования битовых операций приведен в исходном коде макроса U32TO32 (файл macro.h), который в зависимости от порядка байтов, выполняет заданное преобразование. Исходный код макроса приведен в листинге 4.

Листинг 4 – Исходный код макроса U32TO32

#if \_\_BYTE\_OORDER == \_\_BIG\_ENDIAN

#define U32TO32(x)

((x<<24) | ((x<<8) & (0xFF0000) | ((x>>8) & 0xFF00) | (x>>24))

#elif \_\_BYTE\_ORDER == \_\_LITTLE\_ENDIAN

#define U32TO32(x) (x)

#else

#error unsupported byte order

#endif

В приведенном выше макросе используются битовые сдвиги, побитовые умножения и сложения, которые производят поворот машинного слова. Подобная конструкция практически не затрачивает время на поворот машинного слова в виду того, что побитовые операции являются операциями низкого уровня.

## 2.3 Модернизация алгоритмов

В ходе изучения официальной документации и исходных кодов реализаций криптографических алгоритмов, предложенные их авторами и находящиеся в открытом доступе, были найдены участки кода, изменение которых позволило увеличить производительность алгоритмов.

В алгоритмах Salsa и Sosemanuk была модифицирована функция шифрования/расшифровывания. Было уменьшено количество обращений к входному потоку данных за счет использования 32-битных массивов машинных слов, генерируемых за одну итерацию алгоритма. После модернизации, функция шифрования в алгоритме Salsa способна за один вызов обработать до 512 бит данных, вместо 64-х бит, а в алгоритме Sosemanuk – до 640 бит за один проход, вместо 80 проходов по 8 бит. Общая схема модернизации для алгоритма Salsa приведена в листинге 5.

Листинг 5 – Исходный код функции шифрования алгоритма Salsa

uint32\_t keystream[16];

for(; buflen >= 64, buflen -= 64, buf += 64, out += 64) {

salsa20(ctx, keystream);

\*(uint32\_t\*)(out+0) = \*(uint32\_t\*)(buf+0) ^ keystream[0];

\*(uint32\_t\*)(out+4) = \*(uint32\_t\*)(buf+0) ^ keystream[1];

.

.

.

\*(uint32\_t\*)(out+60) = \*(uint32\_t\*)(buf+0) ^ keystream[16];

}

Исходный код функции шифрования алгоритма Sosemanuk аналогичен вышеприведенному, лишь с изменением размера массива keystream.

В алгоритме Trivium разработчики предложили использовать один макрос для инициализации начального состояния алгоритма и генерации ключевой последовательности. Ими не был учтен тот факт, что ресурсы процессора впустую затрачиваются для вычисления 32-х битов ключевой последовательности при инициализации алгоритма (таких пустых вычислений набирается 36). Так же разработчики предложили вариант 2-х макросов для загрузки и выгрузки битов секретного ключа и вектора инициализации при каждой итерации цикла с использованием кроссплатформенных преобразований. Модернизация исходного кода шифра Trivium заключается в исключении пустых вычислений при инициализации начального состояния алгоритма и замены макросов выгрузки битов, на стандартную функцию языка программирования С – memset. Достижение кроссплатформенности достигается при загрузке ключа и вектора инициализации в структуру данных. В листинге 6 представлены модернизированные макросы для инициализации начального состояния и генерации ключевой последовательности.

Листинг 6 – макрос WORK\_1 предназначен для инициализации начального состояния,а WORK\_2 – для генерации ключевой последовательности

#define WORK\_1 { #define WORK\_2{

uint32\_t t1, t2, t3; uint32\_t t1, t2, t3;

T(w); T(w);

UPDATE(w); } z = t1 ^ t2 ^ t3;

UPDATE(w); }

Наибольший выигрыш в скорости удалось добиться в алгоритме Grain путем применения для вычисления битов ключевой последовательности макросов, работающих с массивами регистров не через взаимодействие со структурой данных, а на прямую с битами регистров. Исходный код макросов приведен в листинге 7.

Листинг 7 – макрос LFSR предназначен для генерации нового бита линейного регистра, макрос NFSR – для генерации нового бита нелинейного регистра, макрос OUTBIT – для генерации нового бита ключевой последовательности

#define LFSR(S) (s[0] ^ s[7] ^ s[38] ^ s[70] ^ s[81] ^ s[96])

#define NFSR(b, s) \

(s[0] ^ b[26] ^ b[56] ^ b[91] ^ b[96] ^ \

(b[3] & b[57]) ^ (b[11]) & (b[13]) ^ (b[17] & b[18]) ^ \

(b[27] & (b[59]) ^ (b[40] & b[48]) ^ (b[61] & b[65]) ^ \

(b[68] & b[84]))

#define OUTBIT(b, s) \

(b[2] ^ b[15] ^ b[36] ^ b[45] ^ b[64] ^ b[73] ^ b[89] ^ H(b, s) ^ s[93])

Подобный подход позволил увеличить производительность алгоритма Grain более чем в 2 раза. Результаты тестов будут представлены в следующих разделах.

Также во всех алгоритмах проекта eStream не была заложена проверка на длину вводимых секретного ключа и вектора инициализации. Исходный код проверки для алгоритма Salsa приведен в листинге 8.

Листинг 8 – Процедура проверки длины секретного ключа и вектора инициализации для алгоритма Salsa

if((keylen <= 32) && (keylen > 0))

ctx->keylen = keylen;

else

return -1;

if((ivlen > 0) && (ivlen <= 8))

ctx->ivlen = ivlen;

else

return -1;

Для остальных алгоритмов библиотеки estream.h процедура проверки аналогична, за тем исключением, что сравнение производится с максимально возможной длиной секретного ключа и вектора инициализации предъявляемой к каждому алгоритму.

## 2.4 Сравнение библиотеки estream.h и алгоритмов разработчиков

Использование механизмов модернизации в библиотеке estream.h позволили увеличить скоростные показатели некоторых шифров, по сравнению с реализациями, предложенными авторами алгоритмов. Исходные коды разработчиков были взяты с официального сайта проекта eStream – <http://www.ecrypt.eu.org/stream/>.

Время, затраченное алгоритмами на процесс шифрования информации, будет вычисляться с помощью UNIX-утилиты time. Утилита time производит 3 измерения одновременно: real (реально затраченное пользовательское время), user (процессорное время, затраченное процессом, без учета времени вызова системных функций), sys (процессорное время, затраченное процессом для вызова системных функций). Более точное вычисление реального процессорного времени, которое будет затрачено для шифрования потока данных, вычисляет режим измерения user. В данном режиме не учитывается время, которое затрачивает процесс на ожидание своей очереди исполнения в процессоре и время, затраченное на вызов системных функций.

Измерение скорости шифрования производится на процессоре Intel Atom N2600 Dual Core с тактовой частотой 1.6 GHz (гигагерц). В качестве единицы измерения времени используются миллисекунды (мс), так как они наиболее точно отражают процессорное время, затраченное на процесс шифрования данных.

Для увеличения восприимчивости таблиц с измерениями, введены следующие понятия:

1. estream.h – разработанная в ходе выполнения выпускной квалификационной работы библиотека на основе проекта eStream;
2. eStream – программные реализации алгоритмов, предложенные разработчиками для участия в проекте eStream.

Измерения производятся на следующих объемах данных: 0,5 МБ, 10 МБ и 1 ГБ.

Погрешность при вычислениях вычисляется по следующей формуле:

Формула.png,

где n – число измерений;

– среднее значение;

i – значение i-го измерения;

Sx – значение погрешности.

В таблице 1 приведены результаты тестов на объеме данных 0,5 МБ (524 288 байт).

Таблица 1 – Результаты тестов на объеме данных 0,5 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Grain | Mickey | Trivium |
| 1 | estream.h | 8 | 8 | 2 | 4 | 964 | 300 | 12 |
| eStream | 12 | 7 | 2 | 10 | 2 640 | 300 | 12 |
| 2 | estream.h | 10 | 8 | 2 | 4 | 1 024 | 300 | 8 |
| eStream | 8 | 8 | 2 | 6 | 2 744 | 304 | 12 |
| 3 | estream.h | 8 | 8 | 2 | 4 | 916 | 304 | 8 |
| eStream | 12 | 9 | 2 | 8 | 2 652 | 304 | 12 |
| 4 | estream.h | 12 | 8 | 2 | 4 | 904 | 304 | 12 |
| eStream | 16 | 8 | 2 | 8 | 2 648 | 300 | 12 |
| 5 | estream.h | 8 | 8 | 2 | 6 | 968 | 300 | 8 |
| eStream | 8 | 8 | 2 | 12 | 2 644 | 304 | 12 |
| 6 | estream.h | 12 | 8 | 2 | 4 | 896 | 304 | 12 |
| eStream | 16 | 8 | 2 | 12 | 2 644 | 304 | 12 |
| 7 | estream.h | 8 | 8 | 2 | 8 | 972 | 304 | 4 |
| eStream | 12 | 8 | 2 | 8 | 2 648 | 300 | 8 |
| 8 | estream.h | 8 | 7 | 2 | 8 | 916 | 302 | 8 |
| eStream | 12 | 6 | 2 | 12 | 2 660 | 304 | 8 |
| 9 | estream.h | 8 | 8 | 2 | 8 | 908 | 300 | 8 |
| eStream | 8 | 8 | 2 | 9 | 2 644 | 300 | 8 |
| 10 | estream.h | 8 | 12 | 2 | 4 | 916 | 304 | 12 |
| eStream | 12 | 12 | 2 | 8 | 2 656 | 304 | 12 |
| Среднее значение | estream.h | 9 | 8,3 | 2 | 5,4 | 938,4 | 302,2 | 9,2 |
| eStream | 11,6 | 8,2 | 2 | 9,3 | 2 658 | 302,4 | 10,8 |
| Погрешность измерений | estream.h | 0,5 | 0,4 | 0,1 | 0,6 | 13,1 | 0,6 | 0,9 |
| eStream | 0,9 | 0,3 | 0,1 | 0,7 | 9,7 | 0,6 | 0,6 |

В таблице 2 приведены результаты тестов на объеме данных 10 МБ (10 485 760 байт).

Таблица 2 – Результаты тестов на объеме данных 10 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Grain | Mickey | Trivium |
| 1 | estream.h | 160 | 140 | 48 | 68 | 17 961 | 5 956 | 140 |
| eStream | 200 | 136 | 48 | 108 | 52 691 | 5 976 | 144 |
| 2 | estream.h | 160 | 136 | 44 | 72 | 18 125 | 5 968 | 132 |
| eStream | 200 | 136 | 44 | 100 | 52 579 | 5 964 | 136 |
| 3 | estream.h | 164 | 132 | 48 | 72 | 17 809 | 5 964 | 140 |
| eStream | 204 | 128 | 48 | 100 | 52 459 | 5 972 | 144 |
| 4 | estream.h | 160 | 136 | 48 | 64 | 17 789 | 5 952 | 144 |
| eStream | 196 | 128 | 52 | 104 | 52 551 | 5 964 | 140 |
| 5 | estream.h | 164 | 140 | 52 | 72 | 17 789 | 5 948 | 140 |
| eStream | 196 | 132 | 44 | 100 | 52 555 | 5 968 | 144 |
| 6 | estream.h | 168 | 140 | 48 | 72 | 17 797 | 5 948 | 136 |
| eStream | 200 | 136 | 48 | 100 | 52 547 | 5 972 | 148 |
| 7 | estream.h | 164 | 136 | 48 | 76 | 17 793 | 5 956 | 132 |
| eStream | 192 | 136 | 44 | 100 | 52 539 | 5 968 | 144 |
| 8 | estream.h | 162 | 142 | 48 | 72 | 17 797 | 5 952 | 132 |
| eStream | 200 | 136 | 48 | 112 | 52 547 | 5 972 | 140 |
| 9 | estream.h | 160 | 140 | 48 | 72 | 17 793 | 5 952 | 136 |
| eStream | 200 | 132 | 44 | 104 | 52 535 | 5 968 | 144 |
| 10 | estream.h | 164 | 140 | 44 | 72 | 17 793 | 5 952 | 136 |
| eStream | 200 | 132 | 44 | 112 | 52 543 | 5 952 | 140 |
| Среднее значение | estream.h | 162,6 | 138,2 | 47,6 | 71,2 | 17 844,6 | 5 954,8 | 136,8 |
| eStream | 198,8 | 133,2 | 46,4 | 104 | 52 554,6 | 5 967,6 | 142,4 |
| Погрешность измерений | estream.h | 0,8 | 1,0 | 0,7 | 1,0 | 35,3 | 2,1 | 1,3 |
| eStream | 1,0 | 1,0 | 0,8 | 1,3 | 18,02 | 1,3 | 1,0 |

Исходя из результатов, приведенных в таблицах 1 и 2, можно заключить, что алгоритмы Grain и Mickey, по сравнению с другими шифрами, демонстрируют слабые скоростные показатели. Это связано с тем, что они ориентированы прежде всего на аппаратную реализацию. В связи с этим, данные алгоритмы не будут тестироваться на объеме данных 1ГБ.

В таблице 3 приведены результаты тестов на объеме данных 1 ГБ (1 073 741 824 байт).

Таблица 3 – Результаты тестов на объеме данных 1 ГБ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Trivium |
| 1 | estream.h | 17 181 | 14 617 | 4 604 | 7 248 | 14 105 |
| eStream | 21 057 | 13 949 | 4 632 | 10 941 | 14 917 |
| 2 | estream.h | 17 197 | 14 609 | 4 600 | 7 288 | 14 109 |
| eStream | 21 065 | 13 957 | 4 652 | 10 909 | 14 909 |
| 3 | estream.h | 17 197 | 14 617 | 4 588 | 7 256 | 14 105 |
| eStream | 21 041 | 13 953 | 4 636 | 10 909 | 14 909 |
| 4 | estream.h | 17 213 | 14 609 | 4 596 | 7 240 | 14 105 |
| eStream | 21 049 | 13 949 | 4 628 | 10 897 | 14 929 |
| 5 | estream.h | 17 189 | 14 605 | 4 600 | 7 244 | 14 101 |
| eStream | 21 057 | 13 965 | 4 644 | 10 901 | 14 925 |
| 6 | estream.h | 17 193 | 14 589 | 4 600 | 7 248 | 14 109 |
| eStream | 21 013 | 13 953 | 4 640 | 10 885 | 14 913 |
| 7 | estream.h | 17 217 | 14 565 | 4 596 | 7 244 | 14 101 |
| eStream | 21 013 | 13 973 | 4 640 | 10 913 | 14 925 |
| 8 | estream.h | 17 205 | 14 581 | 4 600 | 7 248 | 14 097 |
| eStream | 21 041 | 13 961 | 4 636 | 10 929 | 14 913 |
| 9 | estream.h | 17 205 | 14 601 | 4 584 | 7 252 | 14 105 |
| eStream | 21 021 | 13 965 | 4 632 | 10 929 | 14 925 |
| 10 | estream.h | 17 193 | 14 585 | 4 624 | 7 252 | 14 109 |
| eStream | 21 045 | 13 957 | 4 636 | 10 973 | 14 913 |
| Среднее значение | estream.h | 17 199 | 14 597,8 | 4 599,2 | 7 252 | 14 104,6 |
| eStream | 21 040,2 | 13 958,2 | 4 637,6 | 10 918,6 | 14 917,8 |
| Погрешность измерений | estream.h | 3,5 | 5,4 | 7,5 | 4,3 | 1,3 |
| eStream | 5,9 | 2,5 | 2,2 | 8,0 | 2,4 |

По результатам, представленным в таблицах 1, 2 и 3, можно заключить следующее:

1. алгоритмы Salsa и Sosemanuk, входящие в состав библиотеки estream.h, с увеличением объема обрабатываемой информации, показывают более быстрые результаты, чем реализации предложенные разработчиками;
2. алгоритм Grain из библиотеки estream.h примерно в 2,9 раза быстрее обрабатывает информацию, чем реализация из проекта eStream;
3. алгоритмы HC128, Trivium и Mickey, входящие в состав библиотеки estream.h, демонстрируют сходные результаты, с реализациями предложенными разработчиками;
4. алгоритм Rabbit из библиотеки estream.h незначительно уступает реализации из проекта eStream, из-за использования разработчиками алгоритма специализированных ассемблерных вставок;
5. алгоритм Trivium из категории «аппаратно ориентированные алгоритмы» имеет эффективную программную реализацию;
6. использование макросов и битовых операций на больших объемах данных приводит к выигрышу по времени;
7. уменьшение количества итераций циклов при шифровании/расшифровывании информации ведет к увеличению производительности.

# ГЛАВА 3. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ БИБЛИОТЕКИ estream.h

## 3.1 Рекомендации разработчикам

Библиотека estream.h предназначена для легкой интеграции с различными проектами разработчиков программного обеспечения. Взаимодействие с библиотекой происходит в несколько этапов:

1. подключение заголовочного файла estream.h к разрабатываемому проекту;
2. инициализация выбранного алгоритма шифрования;
3. ввод секретного ключа и вектора инициализации;
4. шифрование/расшифровывание потока данных, подаваемого на вход алгоритму.

Блок-схема процесса взаимодействия с библиотекой estream.h представлена на рисунке 3.

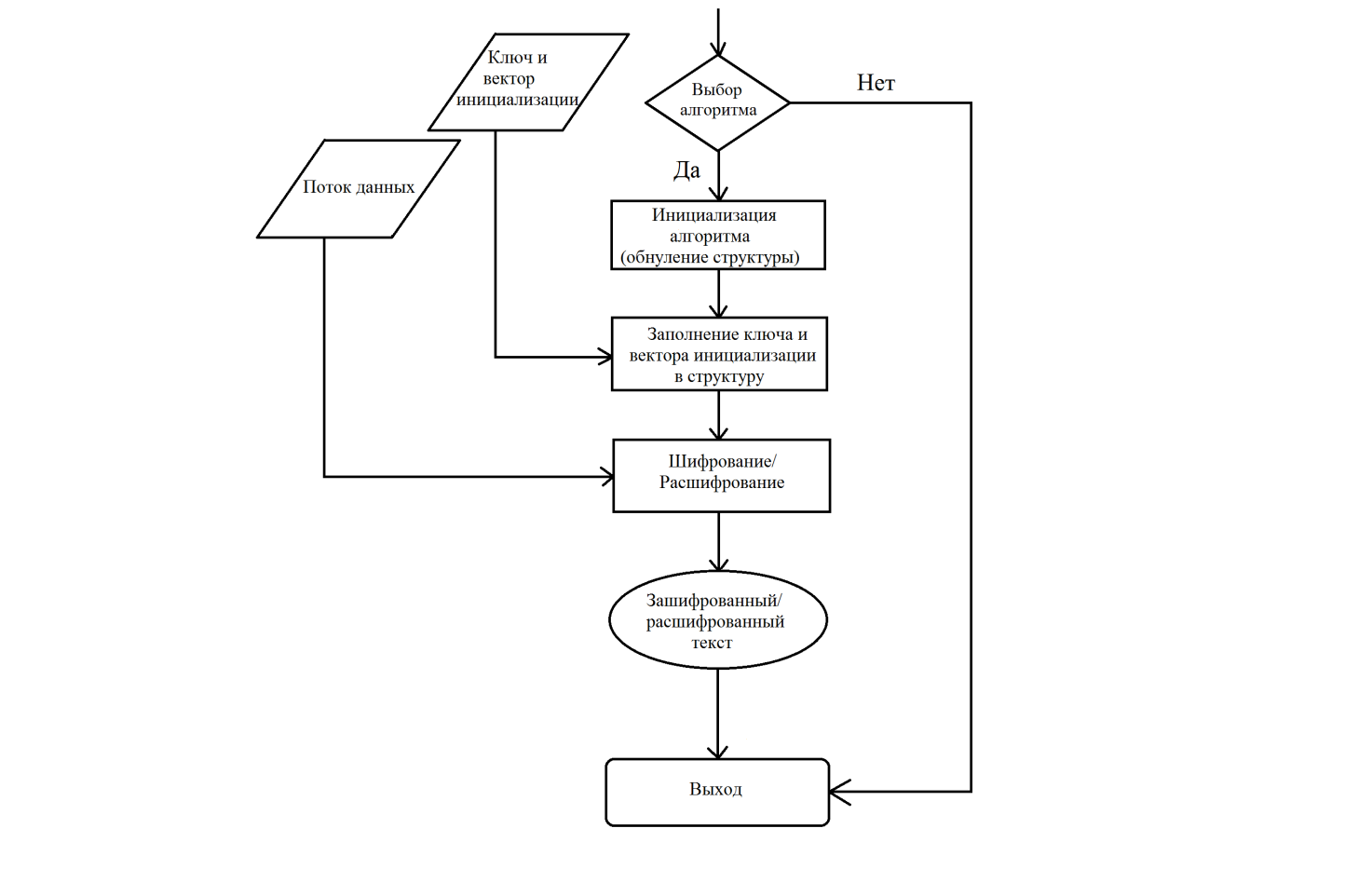


Рисунок 3 – Процесс взаимодействия с библиотекой estream.h

После того как заголовочный файл estream.h будет подключен к проекту, разработчику станут доступны 7 алгоритмов шифрования: Salsa, Rabbit, HC128, Sosemanuk, Grain, Mickey, Trivium.

Выбор алгоритма шифрования осуществляется путем использования названия шифра перед соответствующей функцией. Например, в листинге 9 приведен фрагмент исходного кода, реализующий взаимодействие с библиотекой estream.h, путем использования алгоритма шифрования Sosemanuk.

#define «estream.h» \\ подключение библиотеки

int main()

{

struct sosemanuk\_context ctx; \\ использование структуры данных

if(sosemanuk\_set\_key\_and\_iv(ctx, key, 32, iv, 16))

return -1;

sosemanuk\_crypt(ctx, buf, buflen, buf); \\ функция шифрования

return 0;

}

Листинг 9 – взаимодействие с библиотекой estream.h

Использование других алгоритмов шифрования аналогично вышеприведенному примеру, с изменением первого слова в названиях функций. Количество и содержание аргументов в функциях остается неизменным.

В приложении 1 приведен пример программы ecrypt реализующей взаимодействие со всеми 7-мью алгоритмами шифрования.

## 3.2 Программа для шифрования файлов

Для демонстрации возможностей библиотеки была разработана программа для шифрования/расшифровывания файлов неограниченного объема и различных расширений – ecrypt. Всего было разработано 2 версии программы:

1. с графическим интерфейсом для семейства операционных систем Windows;
2. терминальная версия для операционных систем семейства \*nix.

Графический интерфейс для операционной системы Windows был разработан на платформе Qt Creator с использованием языков программирования C/C++ и QML (специализированный язык программирования платформы Qt).

Графический интерфейс программы изображен на рисунке 4.

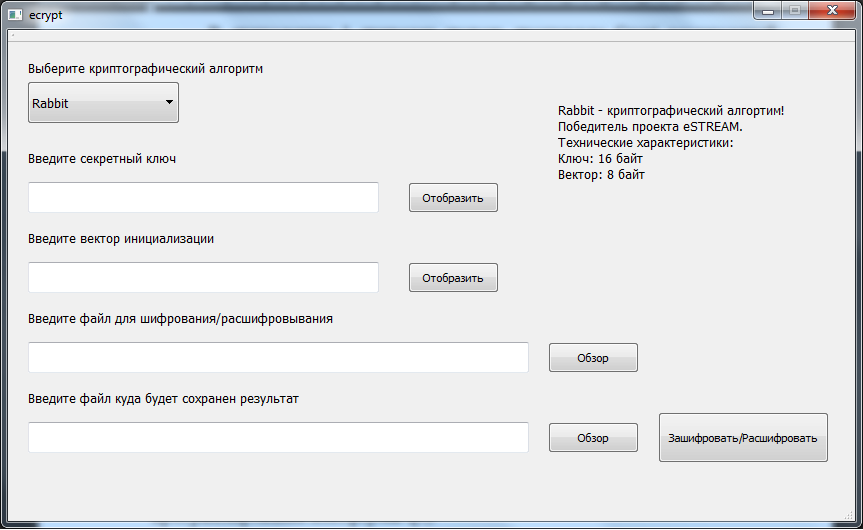


Рисунок 4 – Графический интерфейс программы ecrypt

Как видно из рисунка 4 программа предлагает пользователю ввести секретный ключ, вектор инициализации, выбрать файл для шифрования/расшифровывания и файл для записи результата.

При успешном шифровании/расшифровывании программа предложит пользователю начать новое шифрование. В случае возникновения каких-либо ошибок в ходе выполнения программы, ecrypt оповестит пользователя, выводом ошибки с помощью диалогового окна. На рисунке 5 изображено состояние программы после успешного завершения шифрования.

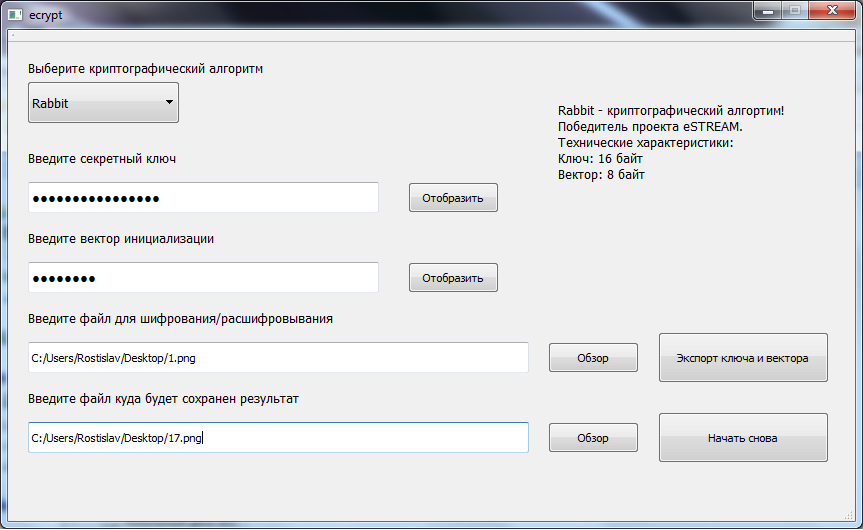


Рисунок 5 – Успешное завершение шифрования/расшифровывания

С исходным кодом программы ecrypt для платформы Windows можно ознакомиться в приложении 2. Исходный код терминальной версии программы ecrypt приведен в приложении 1.

3.3 Клиент-серверное приложение

**Список используемых источников**

<http://www.panasenko.ru/Articles/63/63.html>

http://www.itsec.ru/articles2/Oborandteh/estream\_ditya\_lohnesskogo\_chudovisha

<https://www.ietf.org/proceedings/87/slides/slides-87-tls-2.pdf> - безопасность Salsa

<http://www.itsec.ru/articles2/Oborandteh/estream_bystrye_i_st> - по безопасности estream

<http://www.cryptopp.com/> - библиотека на основе estream

<https://ru.wikipedia.org/wiki/%D0%93%D0%9E%D0%A1%D0%A2_8.417> - гост по оформлению единиц измерения

**ПРИЛОЖЕНИЕ**

Таблица 1 – Результаты измерений на объеме данных 0,5 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Quad Core i7 – 4710HQ с тактовой частотой 2,5 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 1,0 | 1,0 | 1,0 | 1,0 | 71 | 91 | 1,0 |
| eStream | 5,0 | 1,0 | 1,0 | 1,0 | 457 | 90 | 2,0 |
| 2 | estream.h | 1,0 | 1,0 | 1,0 | 1,0 | 67 | 89 | 1,0 |
| eStream | 4,0 | 1,0 | 1,0 | 1,0 | 453 | 89 | 1,0 |
| 3 | estream.h | 1,0 | 2,0 | 2,0 | 1,0 | 65 | 90 | 1,0 |
| eStream | 4,0 | 1,0 | 2,0 | 2,0 | 456 | 85 | 2,0 |
| 4 | estream.h | 1,0 | 1,0 | 1,0 | 1,0 | 71 | 89 | 1,0 |
| eStream | 4,0 | 2,0 | 1,0 | 1,0 | 448 | 90 | 1,0 |
| 5 | estream.h | 3,0 | 1,0 | 2,0 | 1,0 | 71 | 89 | 1,0 |
| eStream | 4,0 | 1,0 | 1,0 | 1,0 | 453 | 85 | 1,0 |
| 6 | estream.h | 3,0 | 1,0 | 1,0 | 2,0 | 69 | 91 | 1,0 |
| eStream | 4,0 | 1,0 | 1,0 | 2,0 | 449 | 86 | 1,0 |
| 7 | estream.h | 3,0 | 2,0 | 1,0 | 1,0 | 72 | 84 | 2,0 |
| eStream | 4,0 | 1,0 | 1,0 | 1,0 | 459 | 85 | 2,0 |
| 8 | estream.h | 3,0 | 1,0 | 1,0 | 2,0 | 72 | 90 | 1,0 |
| eStream | 5,0 | 1,0 | 1,0 | 1,0 | 468 | 90 | 1,0 |
| 9 | estream.h | 3,0 | 2,0 | 1,0 | 1,0 | 71 | 58 | 1,0 |
| eStream | 4,0 | 1,0 | 2,0 | 1,0 | 451 | 59 | 1,0 |
| 10 | estream.h | 1,0 | 1,0 | 1,0 | 2,0 | 73 | 90 | 2,0 |
| eStream | 4,0 | 1,0 | 1,0 | 2,0 | 459 | 89 | 1,0 |
| Среднее значение | estream.h | 2,0 | 1,3 | 1,2 | 1,3 | 70 | 86 | 1,2 |
| eStream | 4,2 | 1,1 | 1,2 | 1,3 | 455 | 85 | 1,3 |
| Погрешность измерений | estream.h | 1,0 | 0,7 | 0,8 | 0,7 | 5 | 5 | 0,8 |
| eStream | 0,8 | 0,9 | 0,8 | 0,7 | 7 | 5 | 0,7 |

Таблица 2 – Результаты измерений на объеме данных 10 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Quad Core i7 – 4710HQ с тактовой частотой 2,5 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 37 | 18 | 9 | 13 | 1 460 | 1 830 | 23 |
| eStream | 56 | 19 | 8 | 18 | 9 740 | 1 820 | 25 |
| 2 | estream.h | 41 | 18 | 8 | 11 | 1 480 | 1 800 | 23 |
| eStream | 60 | 18 | 9 | 18 | 9 750 | 1 810 | 24 |
| 3 | estream.h | 41 | 17 | 8 | 13 | 1 460 | 1 850 | 23 |
| eStream | 59 | 18 | 9 | 22 | 9 700 | 1 810 | 24 |
| 4 | estream.h | 45 | 17 | 9 | 13 | 1 450 | 1 800 | 23 |
| eStream | 56 | 19 | 8 | 22 | 9 740 | 1 780 | 24 |
| 5 | estream.h | 44 | 18 | 8 | 13 | 1 440 | 1 830 | 19 |
| eStream | 59 | 18 | 8 | 22 | 9 750 | 1 800 | 20 |
| 6 | estream.h | 44 | 17 | 8 | 13 | 1 450 | 1 830 | 24 |
| eStream | 60 | 19 | 9 | 18 | 9 730 | 1 810 | 25 |
| 7 | estream.h | 45 | 18 | 8 | 13 | 1 450 | 1 820 | 23 |
| eStream | 59 | 18 | 8 | 22 | 9 700 | 1 810 | 24 |
| 8 | estream.h | 45 | 17 | 8 | 13 | 1 450 | 1 820 | 19 |
| eStream | 56 | 14 | 8 | 22 | 9 750 | 1 810 | 20 |
| 9 | estream.h | 39 | 17 | 8 | 13 | 1 490 | 1 840 | 23 |
| eStream | 59 | 19 | 8 | 18 | 9 750 | 1 800 | 20 |
| 10 | estream.h | 41 | 14 | 9 | 14 | 1 460 | 1 840 | 24 |
| eStream | 63 | 18 | 10 | 22 | 9 710 | 1 780 | 21 |
| Среднее значение | estream.h | 42 | 17 | 8 | 13 | 1 460 | 1 830 | 22 |
| eStream | 59 | 18 | 9 | 20 | 9 730 | 1 800 | 23 |
| Погрешность измерений | estream.h | 5 | 3 | 1 | 2 | 30 | 30 | 3 |
| eStream | 4 | 4 | 1 | 2 | 30 | 30 | 3 |

Таблица 3 – Результаты измерений на объеме данных 1 ГБ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Quad Core i7 – 4710HQ с тактовой частотой 2,5 GHz | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Trivium |
| 1 | estream.h | 4 090 | 1 540 | 660 | 1 100 | 2 120 |
| eStream | 5 560 | 1 570 | 700 | 1 890 | 2 110 |
| 2 | estream.h | 4 100 | 1 580 | 670 | 1 100 | 2 110 |
| eStream | 5 550 | 1 580 | 710 | 1 890 | 2 150 |
| 3 | estream.h | 4 120 | 1 560 | 670 | 1 110 | 2 100 |
| eStream | 5 480 | 1 580 | 700 | 1 880 | 2 140 |
| 4 | estream.h | 4 080 | 1 580 | 660 | 1 100 | 2 090 |
| eStream | 5 550 | 1 570 | 690 | 1 900 | 2 160 |
| 5 | estream.h | 4 120 | 1 580 | 680 | 1 120 | 2 110 |
| eStream | 5 540 | 1 570 | 690 | 1 890 | 2 150 |
| 6 | estream.h | 4 150 | 1 560 | 680 | 1 100 | 2 080 |
| eStream | 5 480 | 1 600 | 690 | 1 880 | 2 130 |
| 7 | estream.h | 4 080 | 1 560 | 670 | 1 090 | 2 120 |
| eStream | 5 530 | 1 580 | 690 | 1 890 | 2 160 |
| 8 | estream.h | 4 070 | 1 570 | 660 | 1 100 | 2 120 |
| eStream | 5 440 | 1 550 | 680 | 1 900 | 2 150 |
| 9 | estream.h | 4 150 | 1 570 | 670 | 1 090 | 2 110 |
| eStream | 5 500 | 1 550 | 690 | 1 880 | 2 170 |
| 10 | estream.h | 4 130 | 1 580 | 680 | 1 100 | 2 110 |
| eStream | 5 540 | 1 600 | 690 | 1 850 | 2 130 |
| Среднее значение | estream.h | 4 110 | 1 570 | 670 | 1 100 | 2 110 |
| eStream | 5 520 | 1 580 | 690 | 1 890 | 2 150 |
| Погрешность измерений | estream.h | 40 | 20 | 10 | 20 | 30 |
| eStream | 40 | 30 | 20 | 20 | 30 |

Таблица 4 – Результаты измерений на объеме данных 0,5 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | AMD Phenom Triple-Core 8450 с тактовой частотой 2,1 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 4,0 | 5,0 | 1,0 | 3,0 | 590 | 176 | 5,0 |
| eStream | 8,0 | 4,0 | 1,0 | 4,0 | 1 120 | 180 | 5,0 |
| 2 | estream.h | 4,0 | 4,0 | 1,0 | 3,0 | 580 | 172 | 8,0 |
| eStream | 12,0 | 4,0 | 2,0 | 4,0 | 1 120 | 160 | 8,0 |
| 3 | estream.h | 8,0 | 4,0 | 1,0 | 4,0 | 570 | 176 | 5,0 |
| eStream | 8,0 | 4,0 | 1,0 | 5,0 | 1 120 | 164 | 8,0 |
| 4 | estream.h | 4,0 | 4,0 | 2,0 | 2,0 | 580 | 176 | 8,0 |
| eStream | 12,0 | 3,0 | 2,0 | 4,0 | 1 130 | 160 | 8,0 |
| 5 | estream.h | 8,0 | 3,0 | 2,0 | 3,0 | 590 | 184 | 5,0 |
| eStream | 8,0 | 5,0 | 1,0 | 4,0 | 1 110 | 176 | 8,0 |
| 6 | estream.h | 8,0 | 4,0 | 2,0 | 3,0 | 570 | 176 | 8,0 |
| eStream | 8,0 | 4,0 | 1,0 | 4,0 | 1 130 | 172 | 8,0 |
| 7 | estream.h | 4,0 | 3,0 | 1,0 | 4,0 | 580 | 176 | 8,0 |
| eStream | 8,0 | 4,0 | 2,0 | 5,0 | 1 120 | 180 | 5,0 |
| 8 | estream.h | 4,0 | 3,0 | 2,0 | 4,0 | 580 | 172 | 5,0 |
| eStream | 8,0 | 3,0 | 1,0 | 4,0 | 1 120 | 168 | 8,0 |
| 9 | estream.h | 8,0 | 4,0 | 2,0 | 3,0 | 590 | 172 | 5,0 |
| eStream | 8,0 | 3,0 | 2,0 | 5,0 | 1 120 | 172 | 8,0 |
| 10 | estream.h | 4,0 | 4,0 | 1,0 | 3,0 | 570 | 168 | 8,0 |
| eStream | 12,0 | 3,0 | 2,0 | 4,0 | 1 140 | 164 | 5,0 |
| Среднее значение | estream.h | 5,6 | 3,8 | 1,5 | 3,2 | 580 | 175 | 6,5 |
| eStream | 9,2 | 3,7 | 1,5 | 4,3 | 1 120 | 170 | 7,1 |
| Погрешность измерений | estream.h | 2,4 | 1,2 | 0,5 | 1,2 | 10 | 9 | 1,5 |
| eStream | 2,8 | 1,3 | 0,5 | 0,7 | 13 | 10 | 2,1 |

Таблица 5 – Результаты измерений на объеме данных 10 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | AMD Phenom Triple-Core 8450 с тактовой частотой 2,1 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 76 | 48 | 28 | 44 | 11 920 | 3 500 | 72 |
| eStream | 96 | 68 | 24 | 52 | 23 500 | 3 430 | 76 |
| 2 | estream.h | 84 | 64 | 20 | 36 | 11 980 | 3 500 | 72 |
| eStream | 104 | 52 | 16 | 56 | 23 560 | 3 420 | 80 |
| 3 | estream.h | 76 | 64 | 16 | 48 | 11 960 | 3 520 | 68 |
| eStream | 100 | 48 | 24 | 48 | 23 460 | 3 450 | 76 |
| 4 | estream.h | 76 | 48 | 16 | 40 | 11 990 | 3 490 | 68 |
| eStream | 88 | 48 | 28 | 52 | 23 440 | 3 440 | 72 |
| 5 | estream.h | 76 | 64 | 28 | 48 | 11 910 | 3 500 | 64 |
| eStream | 112 | 52 | 28 | 52 | 23 430 | 3 430 | 88 |
| 6 | estream.h | 80 | 52 | 20 | 44 | 11 900 | 3 490 | 72 |
| eStream | 100 | 64 | 20 | 56 | 23 500 | 3 410 | 72 |
| 7 | estream.h | 72 | 52 | 28 | 36 | 11 940 | 3 480 | 62 |
| eStream | 96 | 52 | 24 | 52 | 23 470 | 3 420 | 76 |
| 8 | estream.h | 84 | 52 | 28 | 36 | 11 920 | 3 490 | 60 |
| eStream | 100 | 64 | 28 | 48 | 23 470 | 3 420 | 72 |
| 9 | estream.h | 76 | 60 | 20 | 36 | 11 910 | 3 490 | 64 |
| eStream | 96 | 68 | 28 | 52 | 23 460 | 3 430 | 88 |
| 10 | estream.h | 72 | 60 | 20 | 40 | 11 910 | 3 490 | 64 |
| eStream | 100 | 48 | 20 | 60 | 23 490 | 3 420 | 72 |
| Среднее значение | estream.h | 77 | 56 | 22 | 41 | 11 930 | 3 400 | 67 |
| eStream | 99 | 56 | 24 | 53 | 23 480 | 3 430 | 77 |
| Погрешность измерений | estream.h | 7 | 8 | 6 | 7 | 60 | 20 | 7 |
| eStream | 13 | 12 | 8 | 7 | 80 | 30 | 11 |

Таблица 6 – Результаты измерений на объеме данных 1 ГБ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Процессор | AMD Phenom Triple-Core 8450 с тактовой частотой 2,1 GHz | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Trivium |
| 1 | estream.h | 6 580 | 4 880 | 1 620 | 2 950 | 5 790 |
| eStream | 8 220 | 4 730 | 1 600 | 4 230 | 5 950 |
| 2 | estream.h | 6 540 | 4 880 | 1 620 | 2 930 | 5 790 |
| eStream | 8 300 | 4 770 | 1 590 | 4 220 | 5 950 |
| 3 | estream.h | 6 530 | 4 870 | 1 630 | 2 950 | 5 770 |
| eStream | 8 200 | 4 770 | 1 600 | 4 220 | 5 940 |
| 4 | estream.h | 6 530 | 4 880 | 1 620 | 2 940 | 5 800 |
| eStream | 8 300 | 4 760 | 1 580 | 4 190 | 5 960 |
| 5 | estream.h | 6 530 | 4 860 | 1 610 | 2 950 | 5 790 |
| eStream | 8 200 | 4 750 | 1 590 | 4 200 | 5 990 |
| 6 | estream.h | 6 500 | 4 870 | 1 620 | 2 940 | 5 790 |
| eStream | 8 300 | 4 760 | 1 580 | 4 220 | 5 940 |
| 7 | estream.h | 6 560 | 4 870 | 1 620 | 2 970 | 5 790 |
| eStream | 8 300 | 4 740 | 1 580 | 4 220 | 5 940 |
| 8 | estream.h | 6 550 | 4 900 | 1 620 | 2 930 | 5 800 |
| eStream | 8 200 | 4 720 | 1 590 | 4 190 | 5 950 |
| 9 | estream.h | 6 540 | 4 890 | 1 630 | 2 930 | 5 780 |
| eStream | 8 220 | 4 760 | 1 580 | 4 210 | 5 980 |
| 10 | estream.h | 6 560 | 4 860 | 1 630 | 2 920 | 5 800 |
| eStream | 8 290 | 4 750 | 1 600 | 4 220 | 5 950 |
| Среднее значение | estream.h | 6 540 | 4 880 | 1 620 | 2 940 | 5 790 |
| eStream | 8 250 | 4 750 | 1 590 | 4 210 | 5 950 |
| Погрешность измерений | estream.h | 40 | 20 | 10 | 30 | 20 |
| eStream | 50 | 30 | 20 | 20 | 40 |

Таблица 7 – Результаты измерений на объеме данных 0,5 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Atom Dual Core CPU N570 с тактовой частотой 1,66 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 8,0 | 8,0 | 3,0 | 4,0 | 964 | 310 | 12,0 |
| eStream | 12,0 | 7,0 | 3,0 | 10,0 | 2 640 | 300 | 12,0 |
| 2 | estream.h | 10,0 | 8,0 | 2,0 | 4,0 | 975 | 300 | 8,0 |
| eStream | 8,0 | 8,0 | 2,0 | 6,0 | 2 679 | 312 | 12,0 |
| 3 | estream.h | 8,0 | 8,0 | 2,0 | 4,0 | 916 | 304 | 8,0 |
| eStream | 12,0 | 9,0 | 2,0 | 8,0 | 2 652 | 304 | 12,0 |
| 4 | estream.h | 12,0 | 8,0 | 2,0 | 4,0 | 920 | 304 | 12,0 |
| eStream | 16,0 | 8,0 | 2,0 | 8,0 | 2 648 | 300 | 12,0 |
| 5 | estream.h | 8,0 | 8,0 | 2,0 | 6,0 | 968 | 298 | 8,0 |
| eStream | 8,0 | 8,0 | 2,0 | 12,0 | 2 644 | 304 | 12,0 |
| 6 | estream.h | 12,0 | 8,0 | 3,0 | 4,0 | 925 | 304 | 12,0 |
| eStream | 16,0 | 8,0 | 2,0 | 12,0 | 2 644 | 315 | 12,0 |
| 7 | estream.h | 8,0 | 8,0 | 3,0 | 8,0 | 972 | 304 | 6,0 |
| eStream | 12,0 | 8,0 | 2,0 | 8,0 | 2 648 | 300 | 8,0 |
| 8 | estream.h | 8,0 | 7,0 | 2,0 | 8,0 | 916 | 296 | 8,0 |
| eStream | 12,0 | 6,0 | 2,0 | 12,0 | 2 660 | 304 | 8,0 |
| 9 | estream.h | 8,0 | 8,0 | 2,0 | 8,0 | 921 | 300 | 8,0 |
| eStream | 8,0 | 8,0 | 3,0 | 9,0 | 2 644 | 300 | 8,0 |
| 10 | estream.h | 8,0 | 10,0 | 2,0 | 4,0 | 916 | 304 | 12,0 |
| eStream | 12,0 | 10,0 | 3,0 | 8,0 | 2 656 | 304 | 12,0 |
| Среднее значение | estream.h | 9,0 | 8,1 | 2,3 | 5,4 | 940 | 302 | 9,4 |
| eStream | 11,6 | 8,0 | 2,3 | 9,3 | 2 650 | 304 | 10,8 |
| Погрешность измерений | estream.h | 3,0 | 1,9 | 0,7 | 2,6 | 40 | 8 | 3,4 |
| eStream | 4,4 | 2,0 | 0,7 | 3,3 | 30 | 11 | 2,8 |

Таблица 8 – Результаты измерений на объеме данных 10 МБ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Atom Dual Core CPU N570 с тактовой частотой 1,66 GHz | | | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sose-manuk | Grain | Mickey | Trivium |
| 1 | estream.h | 160 | 140 | 48 | 68 | 17 860 | 5 960 | 140 |
| eStream | 200 | 136 | 48 | 108 | 52 700 | 5 980 | 144 |
| 2 | estream.h | 160 | 136 | 44 | 72 | 17 860 | 5 970 | 132 |
| eStream | 200 | 136 | 44 | 100 | 52 600 | 5 960 | 136 |
| 3 | estream.h | 164 | 132 | 48 | 72 | 17 810 | 5 960 | 140 |
| eStream | 204 | 128 | 48 | 100 | 52 500 | 5 970 | 144 |
| 4 | estream.h | 160 | 136 | 48 | 64 | 17 810 | 5 950 | 144 |
| eStream | 196 | 128 | 52 | 104 | 52 600 | 5 960 | 140 |
| 5 | estream.h | 164 | 140 | 52 | 72 | 17 810 | 5 950 | 140 |
| eStream | 196 | 132 | 44 | 100 | 52 600 | 5 970 | 144 |
| 6 | estream.h | 168 | 140 | 48 | 72 | 17 800 | 5 940 | 136 |
| eStream | 200 | 136 | 48 | 100 | 52 600 | 5 970 | 148 |
| 7 | estream.h | 164 | 136 | 48 | 76 | 17 790 | 5 960 | 132 |
| eStream | 192 | 136 | 44 | 100 | 52 500 | 5 970 | 144 |
| 8 | estream.h | 162 | 142 | 48 | 72 | 17 800 | 5 950 | 132 |
| eStream | 200 | 136 | 48 | 112 | 52 600 | 5 970 | 140 |
| 9 | estream.h | 160 | 140 | 48 | 72 | 17 790 | 5 950 | 136 |
| eStream | 200 | 132 | 44 | 104 | 52 500 | 5 970 | 144 |
| 10 | estream.h | 164 | 140 | 44 | 72 | 17 790 | 5 950 | 136 |
| eStream | 200 | 132 | 44 | 112 | 52 500 | 5 950 | 140 |
| Среднее значение | estream.h | 163 | 138 | 48 | 71 | 17 810 | 5 950 | 137 |
| eStream | 199 | 133 | 46 | 104 | 52 600 | 5 970 | 142 |
| Погрешность измерений | estream.h | 5 | 6 | 4 | 7 | 40 | 20 | 7 |
| eStream | 7 | 5 | 6 | 8 | 100 | 20 | 6 |

Таблица 9 – Результаты измерений на объеме данных 1 ГБ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Процессор | Intel Atom Dual Core CPU N570 с тактовой частотой 1,66 GHz | | | | | |
| № п/п теста | Алгоритм  Проект | Salsa | Rabbit | HC128 | Sosemanuk | Trivium |
| 1 | estream.h | 17 180 | 14 620 | 4 600 | 7 250 | 14 110 |
| eStream | 21 060 | 13 950 | 4 630 | 10 940 | 14 920 |
| 2 | estream.h | 17 200 | 14 610 | 4 600 | 7 270 | 14 110 |
| eStream | 21 070 | 13 960 | 4 650 | 10 910 | 14 910 |
| 3 | estream.h | 17 200 | 14 620 | 4 590 | 7 260 | 14 110 |
| eStream | 21 040 | 13 950 | 4 640 | 10 910 | 14 910 |
| 4 | estream.h | 17 210 | 14 610 | 4 600 | 7 240 | 14 110 |
| eStream | 21 050 | 13 950 | 4 630 | 10 900 | 14 930 |
| 5 | estream.h | 17 190 | 14 610 | 4 600 | 7 240 | 14 100 |
| eStream | 21 060 | 13 970 | 4 640 | 10 900 | 14 940 |
| 6 | estream.h | 17 190 | 14 590 | 4 600 | 7 250 | 14 110 |
| eStream | 21 010 | 13 950 | 4 640 | 10 890 | 14 910 |
| 7 | estream.h | 17 220 | 14 570 | 4 600 | 7 240 | 14 100 |
| eStream | 21 010 | 13 970 | 4 640 | 10 910 | 14 930 |
| 8 | estream.h | 17 210 | 14 580 | 4 600 | 7 250 | 14 100 |
| eStream | 21 040 | 13 960 | 4 640 | 10 930 | 14 910 |
| 9 | estream.h | 17 210 | 14 600 | 4 580 | 7 250 | 14 110 |
| eStream | 21 020 | 13 970 | 4 630 | 10 930 | 14 930 |
| 10 | estream.h | 17 190 | 14 590 | 4 620 | 7 250 | 14 120 |
| eStream | 21 050 | 13 960 | 4 640 | 10 950 | 14 910 |
| Среднее значение | estream.h | 17 200 | 14 600 | 4 600 | 7 250 | 14 110 |
| eStream | 21 040 | 13 960 | 4 640 | 10 920 | 14 920 |
| Погрешность измерений | estream.h | 20 | 30 | 30 | 20 | 20 |
| eStream | 30 | 20 | 20 | 30 | 20 |