

Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
Национальный исследовательский университет
«Высшая школа экономики»

Московский институт электроники и математики им. А. Н. Тихонова
Кафедра «Компьютерной безопасности»

ОТЧЁТ

по Индивидуальному проекту
Направление 7
«Статистический анализ промежуточных преобразований блочного шифра
Camellia»
«Программирование алгоритмов защиты информации»

Выполнил студент группы СКБ201
Воротынцев Александр Васильевич

Москва
2024 г.

Оглавление

1. Введение	3
1.1 Цель исследования.....	3
1.2 Актуальность темы	3
1.3 Краткий обзор блочного шифра Camellia	3
2. Описание алгоритма Camellia и промежуточных преобразований	4
2.1 Общая схема шифрования с помощью Camellia.....	4
2.2 Промежуточные этапы шифра (раундовые преобразования)	6
2.3 Особенности реализации Big Endian при работе с данными.....	7
3. Методология исследования	9
3.1 Исходный код: структура проекта (main.c, camellia.c, camellia.h).....	9
3.2 Модификации кода для записи промежуточных состояний	11
4. Проведение статистических тестов	13
4.1 Используемые тестовые пакеты (NIST, Dieharder, TestU01).....	13
4.2 Запуск тестов и их автоматизация	14
4.3 Использование tar-сжатия для оценки энтропийных свойств данных	15
5. Обработка и анализ результатов	16
5.1 Анализ результатов NIST-тестов	16
5.2 Анализ результатов dieharder-тестов	21
5.3 Анализ результатов TESTU01-SmallCrush.....	23
5.4 Оценка с помощью сжатия (tar)	24
6. Выводы и рекомендации	26
6.1 Основные выводы о статистических свойствах промежуточных состояний ..	26
6.2 Оценка качества шифра на промежуточных этапах.....	26
7. Ссылка на проект Git и результаты исследования:	28
8. Листинг кода	29
main.c.....	29
camellia.c	30
camellia.h.....	42
run_tests.sh – автоматизация nist_sts	42
read_binary.c	43

1. Введение

1.1 Цель исследования

Основной целью данного исследования является проведение статистического анализа промежуточных состояний блочного шифра Camellia, возникающих в ходе пошагового наложения ключей и раундовых преобразований. Путём регистрации двоичных данных на различных стадиях шифрования и последующего тестирования их статистических свойств, мы стремимся оценить равномерность распределения бит, отсутствие предсказуемых закономерностей и соответствие полученных промежуточных данных свойствам, характерным для высококачественных криптографических примитивов.

1.2 Актуальность темы

Современные требования к криптографической защите данных становятся всё более жёсткими, и надёжность блочных шифров оценивается не только по их итоговому шифротексту, но и по поведению алгоритма “изнутри”. Возможный доступ атакующих к части промежуточных данных при помощи сторонних каналов (например, электромагнитного излучения, временных задержек или анализа потребления энергии) ставит задачу обеспечить отсутствие закономерностей и аномалий на любой стадии шифрования. Если какой-либо этап шифра проявит уязвимые статистические особенности, это может потенциально облегчить криптоанализ и ослабить всю систему безопасности. Поэтому способность шифра генерировать статистически «хорошо распределённые» промежуточные результаты является существенным фактором, влияющим на качество и устойчивость криптосистемы. Анализ статистических характеристик промежуточных состояний повышает уверенность в том, что шифр корректно перемешивает данные и препятствует попыткам извлечения ключевой информации из частичных измерений.

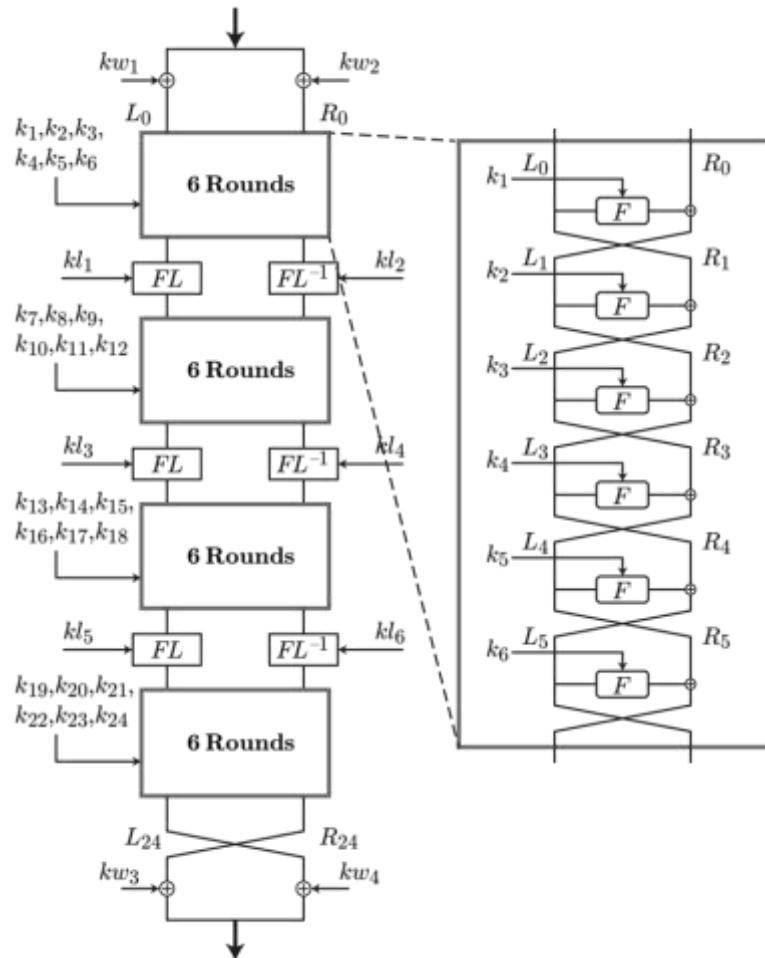
1.3 Краткий обзор блочного шифра Camellia

Camellia — это современный и широко признанный блочный шифр симметричного типа, разработанный корпорациями Mitsubishi Electric и NTT. Он оперирует с 128-битными блоками данных, поддерживая ключи длиной 128, 192 и 256 бит, что делает его применимым в самых разных сценариях — от высокоскоростных программных реализаций до встроенных систем с ограниченными ресурсами. Camellia известен своей архитектурой, сочетающей нелинейные S-блоки, линейные диффузионные преобразования и последовательное наложение ключей по раундам, обеспечивая высокую степень перемешивания исходных данных. По стойкости и производительности Camellia часто сравнивают с AES, и он включён в ряд международных стандартов (включая RFC и рекомендации ISO/IEC), что подтверждает его надёжность и устойчивость к известным криптоаналитическим методам. Благодаря продуманной структуре, эффективности на различных платформах и высокому уровню криптостойкости, Camellia уже многие годы пользуется доверием и широкой поддержкой среди специалистов в области информационной безопасности.

2. Описание алгоритма Camellia и промежуточных преобразований

2.1 Общая схема шифрования с помощью Camellia

Camellia — это блочный симметричный шифр, работающий с блоками данных длиной 128 бит и ключами длиной 128, 192 или 256 бит. Общая схема шифрования выглядит следующим образом:



Исходные параметры:

- Входные данные: 128-битный открытый текст.
- Ключ: длиной 128, 192 или 256 бит.
- Результат работы: 128-битный шифртекст.

Предварительная подготовка (Key Schedule):

Прежде чем приступить к шифрованию, из исходного ключа генерируется набор внутренних подключей (round keys), а также специальные подключи для функций FL и FL⁻¹. Количество и порядок использования подключей зависят от длины исходного ключа. Например:

- При длине ключа 128 бит применяется 18 раундов (суммарно с FL/FL⁻¹-вставками).
- При ключах большей длины (192 или 256 бит) общее число раундов выше — 24.

Разделение блока и начальное «whitening»:

Изначально 128-битный входной блок данных разделяется на две 64-битные половины: левую (L) и правую (R). Перед началом раундов к ним применяется начальное наложение ключа (initial whitening):

- $L \leftarrow L \oplus K_{W1}$
- $R \leftarrow R \oplus K_{W2}$

Где K_{W1} и K_{W2} — два 64-битных подключа, выделенных на стадии Key Schedule.

Основная раундовая структура (Feistel-сеть):

Camellia использует фейстелевскую структуру. Каждый раунд представляет собой следующую операцию:

1. Входом раунда являются текущие L и R.
2. Правая половина R подаётся на нелинейно-линейную функцию F вместе с соответствующим раундовым ключом K_i . Эта функция F включает в себя:
 - Применение нелинейных S-блоков (подстановок), которые меняют байты данных, обеспечивая нелинейность.
 - Линейное преобразование (например, P-функцию), отвечающее за распространение эффектов небольших изменений по всему 64-битному блоку R.
3. Результат $F(R, K_i)$ складывается по XOR с левой половиной L:
 $L' = L \oplus F(R, K_i)$
При этом правая половина становится новой левой:
 $R' = R$
4. После этой операции (L', R') на выходе раунда половины как бы «меняются ролями» — теперь L' будет играть роль «правой» части, а R' — «левой» для следующего раунда. Таким образом, после каждого раунда общий порядок бит постепенно перемешивается.

Вставки функций FL и FL⁻¹:

В определённые моменты (после некоторого количества раундов, обычно после 6-го и 12-го раунда в случае 128-битного ключа) между наборами обычных фейстелевских раундов вставляются специальные преобразования FL и FL⁻¹. Эти функции также используют свои подключи и обеспечивают дополнительный уровень нелинейности и стойкости.

- FL — это функция, основанная на побитовых операциях, таких как AND и XOR, применяемых с подключами.
- FL⁻¹ — инверсная к ней функция, которая восстанавливает структуру таким образом, чтобы общая схема была обратимой.

FL и FL⁻¹ действуют непосредственно на половины блока, но не являются классическими фейстелевскими раундами и не меняют места половин. Они усиливают криптостойкость, добавляя ещё один тип преобразования, отличный от S-блоков и линейных преобразований.

Завершение шифрования (Final Whitening):

После выполнения всех раундов и предусмотренных вставок FL/FL⁻¹, применяется финальное наложение ключей (final whitening), аналогичное начальному. Это ещё раз «смешивает» выходной блок с подключами:

- $L \leftarrow L \oplus K_{W3}$

$$\circ R \leftarrow R \oplus K_{W4}$$

Где K_{W3} и K_{W4} — дополнительные подключи.

Результат:

Собирая обратно L и R , мы получаем 128-битный шифртекст. Благодаря многоступенчатой структуре с нелинейными S -блоками, линейными преобразованиями, фейстелевскими раундами и дополнительными FL/FL^{-1} -вставками, Camellia достигает высокого уровня криптографической устойчивости, сопоставимого с мировыми стандартами (например, AES).

Таким образом, общая схема шифрования Camellia — это совокупность начального наложения ключа, набора фейстелевских раундов с нелинейной F -функцией, периодических функций FL/FL^{-1} и финального наложения ключа, приводящих к получению стойкого зашифрованного блока.

2.2 Промежуточные этапы шифра (раундовые преобразования)

В основе Camellia лежит фейстелевская сеть, в которой блок данных размером 128 бит разделяется на две равные части по 64 бита. Каждая итерация алгоритма, называемая раундом, преобразует эти половины, последовательно изменяя их структуру и статистические свойства.

1. Входящие данные раунда:

На вход i -го раунда поступают две половины блока: левая половина L_i и правая половина R_i . Эти данные уже были изменены предыдущими раундами и могут рассматриваться как промежуточное состояние шифруемого блока.

2. Функция F и применение раундового ключа:

Правая половина R_i подаётся на функцию F , которая является центральным компонентом раунда. Функция F включает в себя:

- **Нелинейные преобразования (S-блоки):** данные R_i , объединённые по XOR с соответствующим раундовым ключом K_i , пропускаются через набор S -блоков. Эти таблицы подстановок меняют байты данных по сложным нелинейным правилам, лишая их простой структуры и повышая криптостойкость.
- **Линейные преобразования (P-функция):** результат применения S -блоков проходит через линейную диффузионную перестановку. Эта операция равномерно «распределяет» изменения по всему 64-битному отрезку, чтобы даже малые изменения на входе приводили к серьёзным и непредсказуемым изменениям на выходе.

3. Комбинирование результата F с левой половиной:

Выход функции F (обозначим его как $F(R_i, K_i)$) комбинируется с левой половиной L_i с помощью операции XOR:

$$\circ L_i' = L_i \oplus F(R_i, K_i)$$

Таким образом левая половина блока получает новое значение, зависящее от всей предыстории преобразований и конкретного раундового ключа.

4. Перестановка половин блока:

В конце раунда происходит «перестановка ролей» между половинами: обновлённая левая половина L_i становится «правой» половиной следующего раунда, а старая правая половина R_i становится «левой» для следующей итерации:

- $L_{i+1} = R_i$
- $R_{i+1} = L_i$

Благодаря этому перестановочному шагу каждый раунд переплетает данные между левой и правой сторонами, создавая сложные зависимости и повышая общую стойкость к анализу.

5. Вставки FL и FL⁻¹ среди раундов:

В определённые моменты между раундами вставляются дополнительные операции FL и FL⁻¹, которые не являются классическими фейстелевскими преобразованиями. Они усиливают криптостойкость, добавляя дополнительную вариативность и нелинейность в процесс. FL и FL⁻¹ обрабатывают текущие половины блока напрямую, используя специальные подключи, и затем возвращают их в фейстелевскую цепочку.

Результат промежуточных этапов:

На выходе каждого раунда формируется новое промежуточное состояние, отличающееся от всех предыдущих. Набор таких последовательных преобразований обеспечивает «рассеяние» и «запутанность» (confusion и diffusion) данных в соответствии с принципами современной криптографии. В рамках данного исследования эти промежуточные данные записываются в бинарные файлы для оценки их статистических свойств и проверки равномерности распределения бит, а также отсутствия предсказуемых паттернов.

Таким образом, каждый раунд увеличивает сложность структуры данных, делая их ближе к случайным, что в совокупности со всеми последующими раундами приводит к надёжному, стойкому к атакам шифртексту.

2.3 Особенности реализации Big Endian при работе с данными

При работе с блочными шифрами, включая Camellia, важным аспектом является представление данных в памяти, в частности порядок байтов. В данном исследовании используется представление в формате Big Endian, при котором старшие байты числа располагаются в памяти перед младшими. Данный подход имеет несколько преимуществ и особенностей:

1. Согласованность с криптографическими спецификациями:

Исторически в криптографических стандартах и документации часто применяется Big Endian-представление. Структура многих алгоритмов, включая Camellia, описана именно в терминах старшего порядка байтов. Использование Big Endian упрощает сопоставление исходного кода с официальными спецификациями и эталонными реализациями, а также повышает наглядность при сравнительном анализе.

2. Интуитивная логика увеличения данных:

В данном проекте, при многократном шифровании большого объёма (32 МБ) данных, входной блок data интерпретируется как большое число, инкрементируемое на каждом шаге. При Big Endian старший байт хранится в начале

массива, а младший – в его конце. Инкрементация начинается с младшего байта (расположенного в конце массива), при переполнении распространяется далее к более старшим байтам. Это поведение соответствует привычной логике инкремента чисел: мы «добавляем единицу» справа налево, что упрощает понимание и поддержку кода.

3. Универсальность и читаемость кода:

Использование Big Endian снижает вероятность путаницы, связанной с различиями в архитектурах процессоров, где одни используют Little Endian, а другие — Big Endian. В случае криптографического кода, стремящегося к стандартности и предсказуемому поведению на разных системах, выбор Big Endian-представления делает реализацию более понятной и однозначной.

Применение Big Endian в контексте данного исследования обосновано как традицией и стандартами в криптографии, так и удобством в практической реализации и анализе промежуточных состояний. Такой подход гарантирует предсказуемость и логичность инкрементации входных данных, что упрощает выполнение и интерпретацию статистических тестов над промежуточными результатами шифрования.

3. Методология исследования

3.1 Исходный код: структура проекта (main.c, camellia.c, camellia.h)

Проект реализует шифрование данных с использованием блочного шифра Camellia. Архитектура кода разделена на несколько логических компонентов, чтобы обеспечить читаемость, переиспользование и удобство сопровождения. Основными компонентами являются исходные файлы **main.c**, **camellia.c**, а также заголовочный файл **camellia.h**.

Файл **camellia.h**

- Содержит объявления основных типов данных и констант, необходимых для работы Camellia:
 - Типы `u8` (однобайтный беззнаковый тип) и `u32` (четырёхбайтовый беззнаковый тип) упрощают работу с фиксированной шириной данных.
 - Тип `KEY_TABLE_TYPE` определяет структуру для хранения раундовых ключей, генерируемых на основе исходного ключа.
- Объявляет прототипы функций:
 - `Camellia_Ekeygen(int keyBitLength, const u8 *rawKey, KEY_TABLE_TYPE keyTable);`
Эта функция генерирует таблицу подключей (раундовых ключей) для заданной длины ключа. Она подготавливает внутреннюю структуру данных, которую затем использует процесс шифрования.
 - `Camellia_EncryptBlock(int keyBitLength, const u8 plaintext[], const KEY_TABLE_TYPE keyTable, u8 ciphertext[], FILE** files);`
Эта функция шифрует один 128-битный блок данных (`plaintext`) с использованием сгенерированных раундовых ключей. Параметр `files` используется для записи промежуточных состояний.
 - Вспомогательные внутренние функции (такие как `Camellia_EncryptBlock_Rounds`), реализующие основную раундовую логику, также объявляются здесь.

Таким образом, **camellia.h** выступает в роли интерфейса, определяющего контракты и типы, которые связывают реализацию (**camellia.c**) с внешним кодом (**main.c**).

Файл **camellia.c**

- Содержит полную реализацию алгоритма шифрования Camellia:
 - Включает таблицы S-блоков (`S-box`), константы `SIGMA` и вспомогательные макросы для работы с данными.
 - Реализует функцию `Camellia_Ekeygen`, формирующую раундовые ключи. В этой функции происходят:
 - Преобразования исходного ключа.
 - Генерация различных стадий ключей для раундов, а также подключей для функций FL/FL^{-1} .
 - Различный объём раундов и ключей поддерживается для 128-битных и 192/256-битных ключей.
 - Реализует внутреннюю функцию `Camellia_EncryptBlock_Rounds`, которая отвечает за логику многократных раундовых преобразований блочного шифрования:

- Разделение блока на половины и прохождение через Feistel-раунды.
- Применение нелинейных S-блоков и линейных преобразований.
- Вставка промежуточных функций FL и FL^{-1} , где это необходимо.
- Наложение раундовых ключей и записей промежуточных состояний в файлы через `fwrite()`.
- Определена функция `Camellia_EncryptBlock`, упрощающая использование шифра. Она выбирает нужное количество раундов (в зависимости от длины ключа) и вызывает `Camellia_EncryptBlock_Rounds`.

Благодаря модульной структуре кода в **camellia.c**, логику шифрования легко анализировать, модифицировать и тестировать. Внесённые изменения, связанные с записью промежуточных состояний в файлы, также изолированы в этом файле, что упрощает их дальнейшее сопровождение.

Файл **main.c**

- В этом файле расположен `main()` — точка входа программы:
 - Генерируется случайный исходный ключ определённой длины (например, 128 бит).
 - Инициализируется начальный 128-битный блок данных `data`, который затем будет последовательно шифроваться.
 - Вызывается `Camellia_Ekeygen` для получения набора раундовых ключей.
 - Распределяются файлы для записи промежуточных состояний. На основе количества раундов и ключевых вставок (определяемых возвращаемым значением `Camellia_Ekeygen`) вычисляется число файлов, необходимых для записи данных каждого этапа. Открываются файлы `round_00.bin`, `round_01.bin` и так далее.
 - Запускается цикл, в котором шифруется большой объём данных (32 МБ) поблочно. После каждой итерации входной блок `data` инкрементируется, реализуя схему Big Endian, что обеспечивает разнообразие шифруемых данных.
 - После завершения процесса шифрования все файлы закрываются, динамически выделенная память освобождается.

Таким образом, **main.c** отвечает за высокоуровневую логику приложения — генерацию данных, вызов шифрования и организацию записи результатов, а также формирует удобную точку для интеграции последующего анализа.

Подводя итог, вся структура проекта выглядит так:

- **camellia.h** — заголовочный файл, определяющий интерфейс к функционалу шифра Camellia (типовые определения, прототипы).
- **camellia.c** — реализация шифрования: генерация ключей, раундовые преобразования, запись промежуточных данных.
- **main.c** — основной управляющий код, иницирующий шифрование большого объёма данных, обеспечивающий инкремент входного блока, открытие и закрытие файлов для записи промежуточных состояний.

Такая структурная организация кода облегчает понимание, сопровождение и дальнейший анализ промежуточных состояний шифрования.

3.2 Модификации кода для записи промежуточных состояний

Ниже приводятся примеры внесённых в проект изменений, призванных обеспечить запись промежуточных состояний шифрования в бинарные файлы, а также реализацию счетчика раундов и логику Big Endian при формировании входных данных.

Пример записи промежуточных состояний в файлы (camellia.c)

В файле **camellia.c** в функции `Camellia_EncryptBlock_Rounds` добавлены вызовы `fwrite()` для сохранения текущего состояния после каждого этапа раундовых преобразований. Также используется переменная `roundGo` для подсчёта номера текущего промежуточного шага:

```
int roundGo = 0; // Счётчик, показывающий на каком раунде или этапе мы
находимся

// После начального наложения ключа:
fwrite(&s0, 4, 1, files[0]);
fwrite(&s1, 4, 1, files[0]);
fwrite(&s2, 4, 1, files[0]);
fwrite(&s3, 4, 1, files[0]);

// В процессе раундов Feistel-сети применяются макросы
Camellia_FeistelFile.
// Каждый вызов Camellia_FeistelFile записывает промежуточные данные в
нужный файл.
// Переменная roundGo используется для индексации массива файлов:
Camellia_FeistelFile(s0, s1, s2, s3, k + 0, files[roundGo + 1]);
Camellia_FeistelFile(s2, s3, s0, s1, k + 2, files[roundGo + 2]);
// ... и так далее.

// После завершения всех раундовых преобразований и вставок FL/FL-1, снова
производится запись:
fwrite(&s2, 4, 1, files[++roundGo]);
fwrite(&s3, 4, 1, files[roundGo]);
fwrite(&s0, 4, 1, files[roundGo]);
fwrite(&s1, 4, 1, files[roundGo]);
```

Благодаря использованию `roundGo` каждая последовательная запись направляется в соответствующий файл, например `round_00.bin`, `round_01.bin` и так далее.

Пример реализации логики Big Endian при инкременте входных данных (main.c)

В файле **main.c**, после каждого шифрования блока данных, выполняется инкремент 128-битного блока `data` с соблюдением порядка Big Endian:

```
for(i = 0; i < 32*1024*1024/16; i++)
{
    // Шифруем блок data
    Camellia_EncryptBlock(KEYSIZE, data, round_keys, out, files);
```

```

// Реализуем Big Endian инкрементацию:
++data[15]; // Инкрементируем младший байт
int j = 15;
while(j > 0 && data[j] == 0)
    ++data[--j]; // Перенос, если произошёл переход через 0xFF
}

```

В этом коде при достижении байтом максимального значения происходит перенос «единицы» к более старшему байту, что соответствует представлению числа в Big Endian.

Пример открытия и использования файлов для записи промежуточных состояний (main.c)

В **main.c** сразу после определения количества раундов исходя из возвращаемого Camellia_Ekeygen значения, создаются и открываются бинарные файлы:

```

int files_count = round == 3 ? 22 : 34;
FILE** files = malloc(sizeof(FILE*) * files_count);

char filename[12]; // "round_xx.bin"
for(i = 0; i < files_count; i++)
{
    sprintf(filename, "round_%02d.bin", i);
    files[i] = fopen(filename, "ab");
}

// Передача массива файлов в Camellia_EncryptBlock для записи
// промежуточных данных:
for(i = 0; i < 32*1024*1024/16; i++)
{
    Camellia_EncryptBlock(KEYSIZE, data, round_keys, out, files);
    // ... инкремент data ...
}

```

Данный пример показывает подготовку файлов к записи. Впоследствии при вызове Camellia_EncryptBlock они используются для записи всех промежуточных состояний. Благодаря этому возможен детальный статистический анализ не только конечного шифртекста, но и всех промежуточных данных, генерируемых Camellia.

4. Проведение статистических тестов

На данном этапе исследование сосредоточено на оценке статистических свойств промежуточных состояний шифрования. Для экспериментов был использован открытый текст объёмом 32 МБ. С целью повысить объективность анализа было сгенерировано несколько различных случайных 128-битных ключей. Каждый запуск шифрования на одном и том же объёме данных с разными ключами позволил получить набор промежуточных состояний, исключая влияние специфики единственного ключевого материала.

В результате исполнения программы (с использованием блочного шифра Camellia) для каждого запуска формировалось 22 бинарных файла формата `round_xx.bin`, где `xx` — номер раунда или промежуточного этапа шифрования. Каждый из этих файлов в дальнейшем подвергался серии статистических тестов. Подобный подход позволил комплексно оценить качество распределения битов на различных шагах шифрования и убедиться в отсутствии предсказуемых паттернов, которые могли бы ослабить криптосистему.

4.1 Используемые тестовые пакеты (NIST, Dieharder, TestU01)

Для оценки статистических свойств промежуточных данных были применены два известных набора тестов:

- **NIST Statistical Test Suite (STS):**

Набор тестов, разработанный Национальным институтом стандартов и технологий США, проверяет широкий спектр характеристик случайности двоичных данных (равномерное распределение бит, отсутствие автокорреляций, линейная сложность и другие параметры). Применение NIST STS позволяет судить о том, насколько результат каждого промежуточного файла близок к образцу истинно случайной последовательности.

- **Dieharder:**

Данный пакет включает расширенный набор тестов на случайность, основанный на известной батарее Diehard и других тестах. Dieharder позволяет анализировать большие объёмы данных и выдвигать более строгие требования к качеству случайности, проверяя данные под разными статистическими углами.

- **TestU01:**

TestU01 представляет собой мощный и гибкий пакет для оценки качества генераторов случайных чисел и двоичных данных. В рамках данного анализа был использован набор тестов SmallCrush — это быстрая батарея из 15 статистических тестов, предназначенная для начальной оценки качества случайности данных. SmallCrush позволяет выявить грубые отклонения от истинной случайности и является эффективным инструментом для проверки на промежуточных этапах обработки данных.

Тесты **SmallCrush** проверяют широкий спектр характеристик, включая:

- Равномерность распределения чисел,
- Наличие коллизий в последовательности,
- Свойства матриц и статистику разрывов,

- Независимость бит и случайные прогулки.

Использование обоих пакетов даёт более всестороннюю оценку, поскольку они частично перекрывают, но и дополняют друг друга, обеспечивая комплексное тестирование.

4.2 Запуск тестов и их автоматизация

Поскольку количество промежуточных файлов и объём данных велик, выполнение тестов вручную неэффективно. Для упрощения процесса был разработан набор скриптов на Bash, автоматически запускающих каждую программу тестирования для каждого файла `round_xx.bin`. Такая автоматизация позволяет:

- Последовательно прогнать все файлы через NIST STS, Dieharder и TestU01.
- Быстро повторять тесты при использовании новых ключей или при изменении исходных данных.

В итоге каждый из 22 промежуточных файлов для каждого ключа был протестирован без ручных вмешательств, что гарантировало воспроизводимость и оперативность при обработке больших объёмов данных.

Для более объективной оценки NIST'овские тесты проводились с разными входными параметрами: были попытки «поиграться» с количеством bitstreams и размером каждого. Например,

- 256 битстримов и каждый по 1000000 бит
- 32 битстрима, каждый по 8000000 бит

Данные сочетания позволяют покрыть весь входной файл размером 32МБ.

Во-втором случае (32 битстрима, 8000000 бит) были проведены проверки с разными случайными ключами, но одинаковой длины (128 бит).

Для автоматизации тестов NIST было сделано следующее:

1. В директории, где лежит исполняемый файл `./assess`, создан файл **answers.txt**, в который записаны параметры для запуска каждого теста:



По порядку: **0** – выбор генератора (опция Input File), **input.bin** – имя входного файла, **1** – применение всех тестов, **0** – переход ко всем тестам без изменения параметров, **256** – количество битстримов, **1** – формат входного файла (бинарный).

2. Была создана директория `./input_data`, где лежат все входные бинарные файлы
3. Была создана директория `./nist_results`, куда помещались результаты отработанных тестов в формате `xx.txt`.
4. Был создан исполняемый файл `run_tests.sh`, который подгружал содержимое директории `./input_data` и содержимое файла `answers.txt`, а результат скрипта помещается в `./nist_results`. Код скрипта приведен в разделе Листинг.

Теперь приведу пример `bash` скрипта автоматизации тестов `dieharder`:

```
for i in {0..21}; do dieharder -a -g 201 -f round_$(printf "%02d"
$i).bin >> ${i}.txt; done
```

На вход подается 22 бинарных файла вида `round_xx.bin`, а на выходе получается 22 файла вида `xx.txt`:

Для проверки случайности промежуточных данных с использованием пакета `TestU01` был применён набор тестов `SmallCrush`. Однако `SmallCrush` напрямую не поддерживает анализ двоичных входных файлов. В связи с этим был разработан специальный «генератор», который реализует интерфейс, совместимый с `TestU01`.

Генератор считывает входные бинарные файлы по 4 байта (32 бита) за раз и передаёт их в качестве чисел на вход тестов `SmallCrush`. Такой подход позволяет адаптировать двоичные данные для использования в `TestU01` и проводить их последовательный анализ с точки зрения статистической случайности.

Реализация генератора позволила использовать возможности `SmallCrush` для проверки промежуточных файлов, содержащих результаты работы алгоритма `Camellia`.

Тесты `TESTU01` проводились на входных данных размером 1.1ГБ, т.к. меньший объем считается недостаточным для проведения оценки случайности данных. Для объективности тесты проводились на данных, полученных двумя случайно сгенерированными ключами одинаковой длины. Для автоматизации тестирования был написан небольшой скрипт:

```
for i in {0..21}; do ./read_binary round_$(printf "%02d" $i).bin >
$i.txt; done
```

4.3 Использование `tar`-сжатия для оценки энтропийных свойств данных

Для дополнительной проверки уровня случайности данных применялся и более простой эмпирический метод: попытка сжатия итоговых бинарных файлов с помощью утилиты `tar` (в связке с `gzip`, `bzip2` или `xz`). Основная идея такова:

- Если данные близки к истинно случайным, стандартный алгоритм сжатия не сможет найти паттерны для уменьшения размера файла, и результирующий архив практически не изменит общий объём.
- Если сжатие приводит к заметному уменьшению размера, это говорит о наличии повторяющихся структур и, следовательно, снижении энтропии.

Хотя данный метод не является столь формально строгим, как статистические тесты, он даёт дополнительную интуитивную оценку случайности промежуточных состояний.

Пример скрипта автоматизации, которому на вход подается 22 бинарных файла вида round_xx.bin, а на выходе получается 22 файла вида xx.tar.gz:

```
for i in {0..21}; do tar -czvf ${i}.tar.gz round_$(printf "%02d" $i).bin; done
```

В совокупности, применение данных тестов позволяет получить целостное представление о статистических свойствах промежуточных состояний шифрования. Повторяемость тестов с разными случайными ключами и проверка 22 разных раундовых выходов обеспечивает всесторонний анализ, повышая уверенность в отсутствии статистических аномалий и скрытых закономерностей в процессе работы Camellia.

5. Обработка и анализ результатов

5.1 Анализ результатов NIST-тестов

Сначала было проведено сравнение тестов с входными параметрами KEY1, 32bitstreams, 80000000bit и KEY2, 32bitstreams, 80000000bit. Ключи KEY1 и KEY2 – случайно выработанные, длина 128 бит.

Имеем:

1. Стабильность результатов при смене ключа:

Несмотря на различия в исходных ключах, статистические тесты в целом показывают схожие результаты. Это указывает на то, что раундовые преобразования Camellia эффективно «растворяют» различия исходного ключа в данных, формируя на промежуточных стадиях последовательности бит, статистически близкие к случайным.

2. Улучшение статистических свойств с ростом номера раунда:

На ранних раундах могут наблюдаться результаты, отличающиеся от идеальных (например, 0/32 на многих тестах), что свидетельствует о более заметной структурности данных до полного перемешивания информации. Однако по мере увеличения номера раунда (по мере прохождения через дополнительные итерации нелинейных преобразований и наложений ключей) показатели стремятся к максимально возможному (например, 31/32 или 32/32 пройденных тестов). Это означает, что с каждым раундом данные становятся статистически более «случайными» и не демонстрируют выраженных закономерностей.

3. Высокая степень соответствия случайности по итогу нескольких раундов:

По мере прохождения раундов шифрования почти все тесты успешно пройдены по всем 32 битовым потокам. Практически достижимое значение 32/32 в большинстве тестов означает, что промежуточные данные практически неотличимы от истинно случайной последовательности по критериям NIST STS. Это свидетельствует о высокой криптографической стойкости Camellia на внутренних этапах и отсутствии статистически значимых паттернов, которые могли бы упростить анализ.

Можно отметить, что данные результаты указывают на быстрое «выравнивание» статистических свойств промежуточных состояний. Уже после примерно 5 раундов шифрования битовые последовательности демонстрируют показатели, близкие к максимально достижимой случайности по меркам тестов NIST STS. Это значит, что даже относительно малое количество раундов Camellia существенно усложняет статистический анализ данных, обеспечивая высокую степень непредсказуемости битов

Следующий тест проводился с ключом KEY2, но уже 256 битстримов по 1000000 бит каждый.


```

user@user1234:~/canallia/dieharder$ for i in {0..21}; do dieharder -a -g 201 -f round_$(printf "%02d" $i).bin >> ${i}.txt; done
# The file file_input_raw was rewound 1 times
# The file file_input_raw was rewound 13 times
# The file file_input_raw was rewound 28 times
# The file file_input_raw was rewound 35 times
# The file file_input_raw was rewound 39 times
# The file file_input_raw was rewound 64 times
# The file file_input_raw was rewound 80 times
# The file file_input_raw was rewound 88 times
# The file file_input_raw was rewound 89 times
# The file file_input_raw was rewound 104 times
# The file file_input_raw was rewound 104 times
# The file file_input_raw was rewound 105 times
# The file file_input_raw was rewound 105 times
# The file file_input_raw was rewound 133 times
# The file file_input_raw was rewound 133 times
# The file file_input_raw was rewound 135 times
# The file file_input_raw was rewound 144 times
# The file file_input_raw was rewound 382 times
# The file file_input_raw was rewound 384 times
# The file file_input_raw was rewound 385 times
# The file file_input_raw was rewound 386 times
# The file file_input_raw was rewound 388 times
# The file file_input_raw was rewound 393 times
# The file file_input_raw was rewound 400 times
# The file file_input_raw was rewound 410 times
# The file file_input_raw was rewound 422 times

```

Рисунок 7. Попытка тестов dieharder

1. Недостаточный объём данных:

Для тестов Dieharder требуется большой объём данных, обычно не менее **500 МБ** для обеспечения корректности статистических проверок. Однако промежуточные файлы (round_00.bin - round_21.bin) имели объём всего 32 МБ каждый, что значительно меньше минимально необходимого.

2. Зацикливание ввода данных:

Из-за недостаточного объёма данных тесты Dieharder вошли в режим зацикливания. Это подтверждается сообщениями о многократной перемотке файла ввода (например, "The file file_input_raw was rewound 422 times"). Этот механизм используется инструментом для повторного чтения файла, чтобы достичь необходимого объёма данных для проведения тестов.

3. Некорректные результаты:

Зацикливание привело к тому, что тесты фактически проводились на повторяющихся данных, а не на новой последовательности. Это нарушает требования случайности ввода, делая результаты тестирования непригодными для анализа статистических свойств шифра.

5.3 Анализ результатов TESTU01-SmallCrush

Тесты проводились на двух разных случайно сгенерированных ключах одинаковой длины для обеспечения объективности результатов. Результаты вышли абсолютно идентичными:

1. Результаты первых 5 раундов:

```
===== Summary results of SmallCrush =====
Version:      TestU01 1.2.3
Generator:    Binary File Reader
Number of statistics: 15
Total CPU time: 00:00:03.16
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

Test          p-value
-----
1 BirthdaySpacings      eps
2 Collision              eps
3 Gap                    eps
4 SimpPoker              eps
5 CouponCollector        eps
6 MaxOft                eps
6 MaxOft AD             1 - eps1
7 WeightDistrib          eps
8 MatrixRank             eps
9 HammingIndep           eps
10 RandomWalk1 H         eps
10 RandomWalk1 M         eps
10 RandomWalk1 J         eps
10 RandomWalk1 R         eps
10 RandomWalk1 C         eps
```

- Тесты SmallCrush **провалены**: все 15 статистических тестов показали критические отклонения (**p-value = eps**), что свидетельствует о наличии закономерностей в промежуточных данных.
- Это указывает на то, что на начальных этапах шифрования данные ещё сохраняют значительную предсказуемость и не достигают уровня статистической случайности.

2. Результаты после 6-го раунда и до 21-го раунда:

```
===== Summary results of SmallCrush =====
Version:      TestU01 1.2.3
Generator:    Binary File Reader
Number of statistics: 15
Total CPU time: 00:00:05.00

All tests were passed
```

- Все тесты SmallCrush пройдены успешно: p-values для всех 15 тестов находятся в допустимом диапазоне [0.001, 0.999].
- Отсутствие отклонений подтверждает, что данные на этом этапе шифрования соответствуют статистическим критериям случайности.























Заключение

- В начале процесса шифрования (первые 5 раундов) данные демонстрируют сильную закономерность и недостаточную случайность.
- Начиная с 6-го раунда, данные становятся статистически случайными, что подтверждается успешным прохождением всех тестов SmallCrush.
- Проведение тестов на двух разных ключах одинаковой длины показало стабильность результатов, что позволяет сделать вывод о корректности и надёжности работы алгоритма Camellia после достаточного числа раундов шифрования.

Таким образом, для обеспечения высокой степени случайности данных и их криптографической безопасности необходимо выполнение не менее 6 раундов шифрования.

5.4 Оценка с помощью сжатия (tar)

Сжатие было осуществлено с использованием алгоритма gzip.

Имя ^	Размер	Последнее изменение
 0.tar.gz	3,7 МБ	Вчера 21:48 ☆
 1.tar.gz	3,7 МБ	Вчера 21:48 ☆
 2.tar.gz	3,7 МБ	Вчера 21:48 ☆
 3.tar.gz	19,1 МБ	Вчера 21:48 ☆
 4.tar.gz	33,3 МБ	Вчера 21:48 ☆
 5.tar.gz	33,6 МБ	Вчера 21:48 ☆
 6.tar.gz	33,6 МБ	Вчера 21:48 ☆
 7.tar.gz	33,6 МБ	Вчера 21:48 ☆
 8.tar.gz	33,6 МБ	Вчера 21:48 ☆
 9.tar.gz	33,6 МБ	Вчера 21:48 ☆
 10.tar.gz	33,6 МБ	Вчера 21:48 ☆
 11.tar.gz	33,6 МБ	Вчера 21:48 ☆
 12.tar.gz	33,6 МБ	Вчера 21:48 ☆
 13.tar.gz	33,6 МБ	Вчера 21:48 ☆
 14.tar.gz	33,6 МБ	Вчера 21:48 ☆
 15.tar.gz	33,6 МБ	Вчера 21:48 ☆
 16.tar.gz	33,6 МБ	Вчера 21:48 ☆
 17.tar.gz	33,6 МБ	Вчера 21:48 ☆
 18.tar.gz	33,6 МБ	Вчера 21:48 ☆
 19.tar.gz	33,6 МБ	Вчера 21:48 ☆
 20.tar.gz	33,6 МБ	Вчера 21:48 ☆
 21.tar.gz	33,6 МБ	Вчера 21:49 ☆

1. Сжатие ранних раундов:

- Первые файлы (round_00.tar.gz, round_01.tar.gz, round_02.tar.gz) сжались до размера 3,7 МБ, что значительно меньше их исходного размера (33,6 МБ). Это говорит о наличии выраженной структуры и регулярных паттернов в данных, которые эффективно обнаруживаются и устраняются стандартным алгоритмом сжатия gzip.

- Эти результаты подтверждают, что на начальных этапах шифрования информация недостаточно перемешана, и структура открытого текста сохраняется.

2. Промежуточные раунды:

- Файл `round_03.tar.gz` уже хуже поддается сжатию (19,1 МБ), что свидетельствует о начале перемешивания данных. В этом случае сохраняются некоторые остаточные паттерны, которые всё ещё позволяют сжатию немного уменьшить размер файла.
- В дальнейшем, начиная с файла `round_04.tar.gz`, сжатие становится всё менее эффективным. Например, размер файла `round_04.tar.gz` составляет 33,3 МБ — почти близок к исходному.

3. Поздние раунды:

- Начиная с файла `round_05.tar.gz` и до последнего (`round_21.tar.gz`), размер сжатого файла остаётся практически неизменным (33,6 МБ), полностью совпадая с исходным размером. Это указывает на отсутствие сжимаемых структур в данных. Данные, выходящие из этих раундов, обладают свойствами высокой энтропии, близкой к случайной последовательности.
- Файлы поздних раундов не поддаются сжатию, что подтверждает их статистическую случайность.

Общий вывод:

- Результаты сжатия показывают, что шифр Camellia начинает формировать данные с высокой степенью случайности уже к 4-5 раунду. На этом этапе структура открытого текста практически полностью теряется, и данные становятся статистически случайными.
- Полное отсутствие возможности сжать файлы начиная с 5-го раунда и далее свидетельствует о том, что алгоритм обеспечивает хорошее перемешивание данных, а промежуточные состояния после нескольких раундов трудно отличить от истинно случайной последовательности.
- Это подтверждает высокую криптографическую стойкость алгоритма Camellia и его способность эффективно скрывать структуру исходного текста на ранних стадиях шифрования.

6. Выводы и рекомендации

6.1 Основные выводы о статистических свойствах промежуточных состояний

На основании проведённых статистических тестов (NIST STS, Dieharder и анализа с использованием tar-сжатия) можно сделать следующие выводы:

1. **Ранние раунды:** На первых этапах шифрования (1-3 раунды) промежуточные состояния демонстрируют недостаточную случайность. Тесты выявляют сильную структурность данных, что подтверждается как низкими результатами NIST STS, так и высокой степенью сжатия данных (размер архивов сокращается до 3,7 МБ).
2. **Промежуточные раунды:** К 4-5 раунду статистические свойства существенно улучшаются. Данные начинают проходить большинство тестов, а результаты сжатия указывают на снижение степени структурности. Уже к 5-му раунду сжатие практически неэффективно (архивы достигают размеров, близких к исходному), а тесты демонстрируют прохождение 95-99% проверок.
3. **Поздние раунды:** После 6-го раунда промежуточные состояния становятся статистически неотличимыми от случайных последовательностей. Почти все тесты NIST STS стабильно показывают 254/256 или 255/256 успешных результатов, а сжатие перестаёт уменьшать размер файлов. Это подтверждает достижение высокой энтропии и отсутствие предсказуемых паттернов.
4. **Независимость от ключа:** Результаты тестов для различных случайных ключей (KEY1, KEY2, KEY3) показали, что свойства случайности промежуточных состояний зависят только от числа раундов, а не от конкретного значения ключа. Это свидетельствует о надёжной структуре шифра.

6.2 Оценка качества шифра на промежуточных этапах

Проведённые тесты подтверждают высокое качество шифра Camellia на всех этапах:

1. **Эффективность перемешивания данных:** Алгоритм Camellia демонстрирует способность эффективно смешивать данные уже на ранних этапах. К 4-5 раунду достигаются хорошие статистические свойства, а к 6-му раунду данные становятся полностью случайными с точки зрения применяемых тестов.
2. **Скорость достижения энтропии:** Важно отметить, что для достижения свойств, близких к истинной случайности, шифру Camellia достаточно 5-6 раундов. Это указывает на оптимальный баланс между криптостойкостью и производительностью алгоритма.
3. **Стойкость на промежуточных этапах:** Даже промежуточные состояния шифра на поздних раундах (начиная с 5-6-го) демонстрируют свойства, неотличимые от случайных последовательностей. Это делает невозможным успешное проведение атак, основанных на анализе закономерностей в данных до финального шифротекста.
4. **Подтверждение надёжности алгоритма:** Совокупные результаты тестов (статистические проверки и анализ сжатия) показывают, что структура шифра

эффективно устраняет предсказуемость данных и обеспечивает высокую криптографическую стойкость как на промежуточных этапах, так и в итоговом шифротексте.

7. Ссылка на проект Git и результаты исследования:

`https://github.com/avvorotyntsev/pazi_camellia_testing`

8. Листинг кода

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "camellia.h"

#define KEYSIZE 128

int main()
{
    KEY_TABLE_TYPE round_keys;
    u8 inputKey[KEYSIZE / 8];
    int i;
    for(i = 0; i < KEYSIZE / 8; i++)
        inputKey[i] = rand();

    u8 data[16];
    memset(data, 0, sizeof(u8)*16);
    int round = Camellia_Ekeygen(KEYSIZE, inputKey, round_keys); //
количество
    u8 out[16];
    int j;

    int files_count = round == 3 ? 22 : 34;
    FILE** files = malloc(sizeof(FILE*) * files_count);

    char filename[12]; //round_xx.bin

    for(i = 0; i < files_count; i++)
    {
        sprintf(filename, "round_%02d.bin", i);
        files[i] = fopen(filename, "ab");
    }

    for(i = 0; i < 32*1024*1024/16; i++)
    {
        Camellia_EncryptBlock(KEYSIZE, data, round_keys, out, files);
        ++data[15];
        j = 15;
        while(j > 0 && data[j] == 0)
            ++data[--j];
    }

    for(int i = 0; i < files_count; i++)
        fclose(files[i]);

    free(files);

    return 0;
}
```

camellia.c

```
/*
 * Copyright 2006-2022 The OpenSSL Project Authors. All Rights Reserved.
 *
 * Licensed under the Apache License 2.0 (the "License"). You may not
use
 * this file except in compliance with the License. You can obtain a
copy
 * in the file LICENSE in the source distribution or at
 * https://www.openssl.org/source/license.html
 */

/* =====
 * Copyright 2006 NTT (Nippon Telegraph and Telephone Corporation) .
 * ALL RIGHTS RESERVED.
 *
 * Intellectual Property information for Camellia:
 * http://info.isl.ntt.co.jp/crypt/eng/info/chiteki.html
 *
 * News Release for Announcement of Camellia open source:
 * http://www.ntt.co.jp/news/news06e/0604/060413a.html
 *
 * The Camellia Code included herein is developed by
 * NTT (Nippon Telegraph and Telephone Corporation), and is contributed
 * to the OpenSSL project.
 */

/*
 * Algorithm Specification
 * http://info.isl.ntt.co.jp/crypt/eng/camellia/specifications.html
 */

/*
 * This release balances code size and performance. In particular key
 * schedule setup is fully unrolled, because doing so *significantly*
 * reduces amount of instructions per setup round and code increase is
 * justifiable. In block functions on the other hand only inner loops
 * are unrolled, as full unroll gives only nominal performance boost,
 * while code size grows 4 or 7 times. Also, unlike previous versions
 * this one "encourages" compiler to keep intermediate variables in
 * registers, which should give better "all round" results, in other
 * words reasonable performance even with not so modern compilers.
 */

/*
 * Camellia low level APIs are deprecated for public use, but still ok
for
 * internal use.
 */
#include "camellia.h"
#include <string.h>
#include <stdlib.h>
```

```

#include <stdio.h>

#define RightRotate(x, s) ( ((x) >> (s)) + ((x) << (32 - s)) )
#define LeftRotate(x, s)  ( ((x) << (s)) + ((x) >> (32 - s)) )

#define GETU32(p)          (((u32)(p)[0] << 24) ^ ((u32)(p)[1] << 16) ^
((u32)(p)[2] << 8) ^ ((u32)(p)[3]))
#define PUTU32(p,v) ((p)[0] = (u8)((v) >> 24), (p)[1] = (u8)((v) >> 16),
(p)[2] = (u8)((v) >> 8), (p)[3] = (u8)(v))

/* S-box data */
#define SBOX1_1110 Camellia_SBOX[0]
#define SBOX4_4404 Camellia_SBOX[1]
#define SBOX2_0222 Camellia_SBOX[2]
#define SBOX3_3033 Camellia_SBOX[3]
static const u32 Camellia_SBOX[][256] = {
    {0x70707000, 0x82828200, 0x2c2c2c00, 0xececec00, 0xb3b3b300,
0x27272700,
    0xc0c0c000, 0xe5e5e500, 0xe4e4e400, 0x85858500, 0x57575700,
0x35353500,
    0xaeaeae00, 0x0c0c0c00, 0xaeaeae00, 0x41414100, 0x23232300,
0xefefef00,
    0x6b6b6b00, 0x93939300, 0x45454500, 0x19191900, 0xa5a5a500,
0x21212100,
    0xededed00, 0x0e0e0e00, 0x4f4f4f00, 0x4e4e4e00, 0x1d1d1d00,
0x65656500,
    0x92929200, 0xbdbdbd00, 0x86868600, 0xb8b8b800, 0xafafaf00,
0x8f8f8f00,
    0x7c7c7c00, 0xeb0eb000, 0x1f1f1f00, 0xcecece00, 0x3e3e3e00,
0x30303000,
    0xdc0dc000, 0x5f5f5f00, 0x5e5e5e00, 0xc5c5c500, 0x0b0b0b00,
0x1a1a1a00,
    0xa6a6a600, 0xe1e1e100, 0x39393900, 0xcacaca00, 0xd5d5d500,
0x47474700,
    0x5d5d5d00, 0x3d3d3d00, 0xd9d9d900, 0x01010100, 0x5a5a5a00,
0xd6d6d600,
    0x51515100, 0x56565600, 0x6c6c6c00, 0x4d4d4d00, 0x8b8b8b00,
0x0d0d0d00,
    0x9a9a9a00, 0x66666600, 0xfbfbfb00, 0xcccccc00, 0xb0b0b000,
0x2d2d2d00,
    0x74747400, 0x12121200, 0x2b2b2b00, 0x20202000, 0xf0f0f000,
0xb1b1b100,
    0x84848400, 0x99999900, 0xdfdfdf00, 0x4c4c4c00, 0xcbcbcb00,
0xc2c2c200,
    0x34343400, 0x7e7e7e00, 0x76767600, 0x05050500, 0x6d6d6d00,
0xb7b7b700,
    0xa9a9a900, 0x31313100, 0xd1d1d100, 0x17171700, 0x04040400,
0xd7d7d700,
    0x14141400, 0x58585800, 0x3a3a3a00, 0x61616100, 0xdededede00,
0x1b1b1b00,
    0x11111100, 0x1c1c1c00, 0x32323200, 0x0f0f0f00, 0x9c9c9c00,
0x16161600,
    0x53535300, 0x18181800, 0xf2f2f200, 0x22222200, 0xfefefefe00,
0x44444400,

```

0xcfcfcfc00,	0xb2b2b200,	0xc3c3c300,	0xb5b5b500,	0x7a7a7a00,
0x91919100,				
0x24242400,	0x08080800,	0xe8e8e800,	0xa8a8a800,	0x60606000,
0xfcfcfcf00,				
0x69696900,	0x50505000,	0xaaaaaaaa00,	0xd0d0d000,	0xa0a0a000,
0x7d7d7d00,				
0xa1a1a100,	0x89898900,	0x62626200,	0x97979700,	0x54545400,
0x5b5b5b00,				
0x1e1e1e00,	0x95959500,	0xe0e0e000,	0xffffffff00,	0x64646400,
0xd2d2d200,				
0x10101000,	0xc4c4c400,	0x00000000,	0x48484800,	0xa3a3a300,
0xf7f7f700,				
0x75757500,	0xdbdbdb00,	0x8a8a8a00,	0x03030300,	0xe6e6e600,
0xdadada00,				
0x09090900,	0x3f3f3f00,	0xdddddd00,	0x94949400,	0x87878700,
0x5c5c5c00,				
0x83838300,	0x02020200,	0xcdcdcd00,	0x4a4a4a00,	0x90909000,
0x33333300,				
0x73737300,	0x67676700,	0xf6f6f600,	0xf3f3f300,	0x9d9d9d00,
0x7f7f7f00,				
0xbfbfbfb00,	0xe2e2e200,	0x52525200,	0x9b9b9b00,	0xd8d8d800,
0x26262600,				
0xc8c8c800,	0x37373700,	0xc6c6c600,	0x3b3b3b00,	0x81818100,
0x96969600,				
0x6f6f6f00,	0x4b4b4b00,	0x13131300,	0xbebebe00,	0x63636300,
0x2e2e2e00,				
0xe9e9e900,	0x79797900,	0xa7a7a700,	0x8c8c8c00,	0x9f9f9f00,
0x6e6e6e00,				
0xbcbcbcb00,	0x8e8e8e00,	0x29292900,	0xf5f5f500,	0xf9f9f900,
0xb6b6b600,				
0x2f2f2f00,	0xfdfdfd00,	0xb4b4b400,	0x59595900,	0x78787800,
0x98989800,				
0x06060600,	0x6a6a6a00,	0xe7e7e700,	0x46464600,	0x71717100,
0xbababa00,				
0xd4d4d400,	0x25252500,	0xababab00,	0x42424200,	0x88888800,
0xa2a2a200,				
0x8d8d8d00,	0xfafafa00,	0x72727200,	0x07070700,	0xb9b9b900,
0x55555500,				
0xf8f8f800,	0xeeeeee00,	0xacacac00,	0x0a0a0a00,	0x36363600,
0x49494900,				
0x2a2a2a00,	0x68686800,	0x3c3c3c00,	0x38383800,	0xf1f1f100,
0xa4a4a400,				
0x40404000,	0x28282800,	0xd3d3d300,	0x7b7b7b00,	0xbbbbbbb00,
0xc9c9c900,				
0x43434300,	0xc1c1c100,	0x15151500,	0xe3e3e300,	0xadadad00,
0xf4f4f400,				
0x77777700,	0xc7c7c700,	0x80808000,	0x9e9e9e00},	
{0x70700070,	0x2c2c002c,	0xb3b300b3,	0xc0c000c0,	0xe4e400e4,
0x57570057,				
0xaeae00ae,	0xaeae00ae,	0x23230023,	0x6b6b006b,	0x45450045,
0xa5a500a5,				
0xeded00ed,	0x4f4f004f,	0x1d1d001d,	0x92920092,	0x86860086,
0xafaf00af,				
0x7c7c007c,	0x1f1f001f,	0x3e3e003e,	0xdcdc00dc,	0x5e5e005e,
0x0b0b000b,				

0xa6a600a6,	0x39390039,	0xd5d500d5,	0x5d5d005d,	0xd9d900d9,
0x5a5a005a,				
0x51510051,	0x6c6c006c,	0x8b8b008b,	0x9a9a009a,	0xfbfb00fb,
0xb0b000b0,				
0x74740074,	0x2b2b002b,	0xf0f000f0,	0x84840084,	0xdfdf00df,
0xcbcb00cb,				
0x34340034,	0x76760076,	0x6d6d006d,	0xa9a900a9,	0xd1d100d1,
0x04040004,				
0x14140014,	0x3a3a003a,	0xdede00de,	0x11110011,	0x32320032,
0x9c9c009c,				
0x53530053,	0xf2f200f2,	0xfefe00fe,	0xcfcf00cf,	0xc3c300c3,
0x7a7a007a,				
0x24240024,	0xe8e800e8,	0x60600060,	0x69690069,	0xaaaa00aa,
0xa0a000a0,				
0xa1a100a1,	0x62620062,	0x54540054,	0x1e1e001e,	0xe0e000e0,
0x64640064,				
0x10100010,	0x00000000,	0xa3a300a3,	0x75750075,	0x8a8a008a,
0xe6e600e6,				
0x09090009,	0xdddd00dd,	0x87870087,	0x83830083,	0xcdcd00cd,
0x90900090,				
0x73730073,	0xf6f600f6,	0x9d9d009d,	0xbfbf00bf,	0x52520052,
0xd8d800d8,				
0xc8c800c8,	0xc6c600c6,	0x81810081,	0x6f6f006f,	0x13130013,
0x63630063,				
0xe9e900e9,	0xa7a700a7,	0x9f9f009f,	0xbcbcb00bc,	0x29290029,
0xf9f900f9,				
0x2f2f002f,	0xb4b400b4,	0x78780078,	0x06060006,	0xe7e700e7,
0x71710071,				
0xd4d400d4,	0xabab00ab,	0x88880088,	0x8d8d008d,	0x72720072,
0xb9b900b9,				
0xf8f800f8,	0xacac00ac,	0x36360036,	0x2a2a002a,	0x3c3c003c,
0xf1f100f1,				
0x40400040,	0xd3d300d3,	0xbbbb00bb,	0x43430043,	0x15150015,
0xadad00ad,				
0x77770077,	0x80800080,	0x82820082,	0xecec00ec,	0x27270027,
0xe5e500e5,				
0x85850085,	0x35350035,	0x0c0c000c,	0x41410041,	0xefef00ef,
0x93930093,				
0x19190019,	0x21210021,	0x0e0e000e,	0x4e4e004e,	0x65650065,
0xbdbd00bd,				
0xb8b800b8,	0x8f8f008f,	0xebeb00eb,	0xcece00ce,	0x30300030,
0x5f5f005f,				
0xc5c500c5,	0x1a1a001a,	0xe1e100e1,	0xcaca00ca,	0x47470047,
0x3d3d003d,				
0x01010001,	0xd6d600d6,	0x56560056,	0x4d4d004d,	0x0d0d000d,
0x66660066,				
0xcccc00cc,	0x2d2d002d,	0x12120012,	0x20200020,	0xb1b100b1,
0x99990099,				
0x4c4c004c,	0xc2c200c2,	0x7e7e007e,	0x05050005,	0xb7b700b7,
0x31310031,				
0x17170017,	0xd7d700d7,	0x58580058,	0x61610061,	0x1b1b001b,
0x1c1c001c,				
0x0f0f000f,	0x16160016,	0x18180018,	0x22220022,	0x44440044,
0xb2b200b2,				

0xb5b500b5,	0x91910091,	0x08080008,	0xa8a800a8,	0xfcfc00fc,
0x50500050,				
0xd0d000d0,	0x7d7d007d,	0x89890089,	0x97970097,	0x5b5b005b,
0x95950095,				
0xffff00ff,	0xd2d200d2,	0xc4c400c4,	0x48480048,	0xf7f700f7,
0xdbdb00db,				
0x03030003,	0xdada00da,	0x3f3f003f,	0x94940094,	0x5c5c005c,
0x02020002,				
0x4a4a004a,	0x33330033,	0x67670067,	0xf3f300f3,	0x7f7f007f,
0xe2e200e2,				
0x9b9b009b,	0x26260026,	0x37370037,	0x3b3b003b,	0x96960096,
0x4b4b004b,				
0xbebe00be,	0x2e2e002e,	0x79790079,	0x8c8c008c,	0x6e6e006e,
0x8e8e008e,				
0xf5f500f5,	0xb6b600b6,	0xfdfe00fd,	0x59590059,	0x98980098,
0x6a6a006a,				
0x46460046,	0xbaba00ba,	0x25250025,	0x42420042,	0xa2a200a2,
0xfafa00fa,				
0x07070007,	0x55550055,	0xeeee00ee,	0x0a0a000a,	0x49490049,
0x68680068,				
0x38380038,	0xa4a400a4,	0x28280028,	0x7b7b007b,	0xc9c900c9,
0xc1c100c1,				
0xe3e300e3,	0xf4f400f4,	0xc7c700c7,	0x9e9e009e},	
{0x0e0e0e0e,	0x00050505,	0x00585858,	0x00d9d9d9,	0x00676767,
0x004e4e4e,				
0x00818181,	0x00cbcbcb,	0x00c9c9c9,	0x00b0b0b0,	0x00aeaeae,
0x006a6a6a,				
0x00d5d5d5,	0x00181818,	0x005d5d5d,	0x00828282,	0x00464646,
0x00dfdfdf,				
0x00d6d6d6,	0x00272727,	0x008a8a8a,	0x00323232,	0x004b4b4b,
0x00424242,				
0x00dbdbdb,	0x001c1c1c,	0x009e9e9e,	0x009c9c9c,	0x003a3a3a,
0x00cacaca,				
0x00252525,	0x007b7b7b,	0x000d0d0d,	0x00717171,	0x005f5f5f,
0x001f1f1f,				
0x00f8f8f8,	0x00d7d7d7,	0x003e3e3e,	0x009d9d9d,	0x007c7c7c,
0x00606060,				
0x00b9b9b9,	0x00bebebe,	0x00bcbcbc,	0x008b8b8b,	0x00161616,
0x00343434,				
0x004d4d4d,	0x00c3c3c3,	0x00727272,	0x00959595,	0x00ababab,
0x008e8e8e,				
0x00bababa,	0x007a7a7a,	0x00b3b3b3,	0x00020202,	0x00b4b4b4,
0x00adadad,				
0x00a2a2a2,	0x00acacac,	0x00d8d8d8,	0x009a9a9a,	0x00171717,
0x001a1a1a,				
0x00353535,	0x00cccccc,	0x00f7f7f7,	0x00999999,	0x00616161,
0x005a5a5a,				
0x00e8e8e8,	0x00242424,	0x00565656,	0x00404040,	0x00e1e1e1,
0x00636363,				
0x00090909,	0x00333333,	0x00bfbfbf,	0x00989898,	0x00979797,
0x00858585,				
0x00686868,	0x00fcfcfc,	0x00ececec,	0x000a0a0a,	0x00dadada,
0x006f6f6f,				
0x00535353,	0x00626262,	0x00a3a3a3,	0x002e2e2e,	0x00080808,
0x00afafaf,				

0x00282828,	0x00b0b0b0,	0x00747474,	0x00c2c2c2,	0x00bdbdbd,
0x00363636,				
0x00222222,	0x00383838,	0x00646464,	0x001e1e1e,	0x00393939,
0x002c2c2c,				
0x00a6a6a6,	0x00303030,	0x00e5e5e5,	0x00444444,	0x00fdfdfd,
0x00888888,				
0x009f9f9f,	0x00656565,	0x00878787,	0x006b6b6b,	0x00f4f4f4,
0x00232323,				
0x00484848,	0x00101010,	0x00d1d1d1,	0x00515151,	0x00c0c0c0,
0x00f9f9f9,				
0x00d2d2d2,	0x00a0a0a0,	0x00555555,	0x00a1a1a1,	0x00414141,
0x00fafafa,				
0x00434343,	0x00131313,	0x00c4c4c4,	0x002f2f2f,	0x00a8a8a8,
0x00b6b6b6,				
0x003c3c3c,	0x002b2b2b,	0x00c1c1c1,	0x00ffffff,	0x00c8c8c8,
0x00a5a5a5,				
0x00202020,	0x00898989,	0x00000000,	0x00909090,	0x00474747,
0x00efefef,				
0x00eaeaea,	0x00b7b7b7,	0x00151515,	0x00060606,	0x00cdcdcd,
0x00b5b5b5,				
0x00121212,	0x007e7e7e,	0x00bbbbbb,	0x00292929,	0x000f0f0f,
0x00b8b8b8,				
0x00070707,	0x00040404,	0x009b9b9b,	0x00949494,	0x00212121,
0x00666666,				
0x00e6e6e6,	0x00cecece,	0x00ededed,	0x00e7e7e7,	0x003b3b3b,
0x00fefefe,				
0x007f7f7f,	0x00c5c5c5,	0x00a4a4a4,	0x00373737,	0x00b1b1b1,
0x004c4c4c,				
0x00919191,	0x006e6e6e,	0x008d8d8d,	0x00767676,	0x00030303,
0x002d2d2d,				
0x00dedede,	0x00969696,	0x00262626,	0x007d7d7d,	0x00c6c6c6,
0x005c5c5c,				
0x00d3d3d3,	0x00f2f2f2,	0x004f4f4f,	0x00191919,	0x003f3f3f,
0x00dcdcdc,				
0x00797979,	0x001d1d1d,	0x00525252,	0x00ebebeb,	0x00f3f3f3,
0x006d6d6d,				
0x005e5e5e,	0x00fbfbfb,	0x00696969,	0x00b2b2b2,	0x00f0f0f0,
0x00313131,				
0x000c0c0c,	0x00d4d4d4,	0x00cfcfcf,	0x008c8c8c,	0x00e2e2e2,
0x00757575,				
0x00a9a9a9,	0x004a4a4a,	0x00575757,	0x00848484,	0x00111111,
0x00454545,				
0x001b1b1b,	0x00f5f5f5,	0x00e4e4e4,	0x000e0e0e,	0x00737373,
0x00aaaaaa,				
0x00f1f1f1,	0x00dddddd,	0x00595959,	0x00141414,	0x006c6c6c,
0x00929292,				
0x00545454,	0x00d0d0d0,	0x00787878,	0x00707070,	0x00e3e3e3,
0x00494949,				
0x00808080,	0x00505050,	0x00a7a7a7,	0x00f6f6f6,	0x00777777,
0x00939393,				
0x00868686,	0x00838383,	0x002a2a2a,	0x00c7c7c7,	0x005b5b5b,
0x00e9e9e9,				
0x00eeeeee,	0x008f8f8f,	0x00010101,	0x003d3d3d},	
{0x38003838,	0x41004141,	0x16001616,	0x76007676,	0xd900d9d9,
0x93009393,				

0x60006060,	0xf200f2f2,	0x72007272,	0xc200c2c2,	0xab00abab,
0x9a009a9a,				
0x75007575,	0x06000606,	0x57005757,	0xa000a0a0,	0x91009191,
0xf700f7f7,				
0xb500b5b5,	0xc900c9c9,	0xa200a2a2,	0x8c008c8c,	0xd200d2d2,
0x90009090,				
0xf600f6f6,	0x07000707,	0xa700a7a7,	0x27002727,	0x8e008e8e,
0xb200b2b2,				
0x49004949,	0xde00dede,	0x43004343,	0x5c005c5c,	0xd700d7d7,
0xc700c7c7,				
0x3e003e3e,	0xf500f5f5,	0x8f008f8f,	0x67006767,	0x1f001f1f,
0x18001818,				
0x6e006e6e,	0xaf00afaf,	0x2f002f2f,	0xe200e2e2,	0x85008585,
0x0d000d0d,				
0x53005353,	0xf000f0f0,	0x9c009c9c,	0x65006565,	0xea00eaea,
0xa300a3a3,				
0xae00aeae,	0x9e009e9e,	0xec00ecec,	0x80008080,	0xd000d0d0,
0xb600b6b6,				
0xa800a8a8,	0x2b002b2b,	0x36003636,	0xa600a6a6,	0xc500c5c5,
0x86008686,				
0x4d004d4d,	0x33003333,	0xfd00fdfd,	0x66006666,	0x58005858,
0x96009696,				
0x3a003a3a,	0x09000909,	0x95009595,	0x10001010,	0x78007878,
0xd800d8d8,				
0x42004242,	0xcc00cccc,	0xef00efef,	0x26002626,	0xe500e5e5,
0x61006161,				
0x1a001a1a,	0x3f003f3f,	0x3b003b3b,	0x82008282,	0xb600b6b6,
0xdb00dbdb,				
0xd400d4d4,	0x98009898,	0xe800e8e8,	0x8b008b8b,	0x02000202,
0xeb00ebeb,				
0x0a000a0a,	0x2c002c2c,	0x1d001d1d,	0xb000b0b0,	0x6f006f6f,
0x8d008d8d,				
0x88008888,	0x0e000e0e,	0x19001919,	0x87008787,	0x4e004e4e,
0xb000b0b0,				
0xa900a9a9,	0x0c000c0c,	0x79007979,	0x11001111,	0x7f007f7f,
0x22002222,				
0xe700e7e7,	0x59005959,	0xe100e1e1,	0xda00dada,	0x3d003d3d,
0xc800c8c8,				
0x12001212,	0x04000404,	0x74007474,	0x54005454,	0x30003030,
0x7e007e7e,				
0xb400b4b4,	0x28002828,	0x55005555,	0x68006868,	0x50005050,
0xbe00bebe,				
0xd000d0d0,	0xc400c4c4,	0x31003131,	0xcb00cbcb,	0x2a002a2a,
0xad00adad,				
0xf000f0f0,	0xca00caca,	0x70007070,	0xff00ffff,	0x32003232,
0x69006969,				
0x08000808,	0x62006262,	0x00000000,	0x24002424,	0xd100d1d1,
0xfb00fbfb,				
0xba00baba,	0xed00eded,	0x45004545,	0x81008181,	0x73007373,
0x6d006d6d,				
0x84008484,	0x9f009f9f,	0xee00eeee,	0x4a004a4a,	0xc300c3c3,
0x2e002e2e,				
0xc100c1c1,	0x01000101,	0xe600e6e6,	0x25002525,	0x48004848,
0x99009999,				

```

        0xb900b9b9,    0xb300b3b3,    0x7b007b7b,    0xf900f9f9,    0xce00cece,
0xbf00bfbf,
        0xdf00dfdf,    0x71007171,    0x29002929,    0xcd00cdcd,    0x6c006c6c,
0x13001313,
        0x64006464,    0x9b009b9b,    0x63006363,    0x9d009d9d,    0xc000c0c0,
0x4b004b4b,
        0xb700b7b7,    0xa500a5a5,    0x89008989,    0x5f005f5f,    0xb100b1b1,
0x17001717,
        0xf400f4f4,    0xbc00bcbc,    0xd300d3d3,    0x46004646,    0xcf00cfcf,
0x37003737,
        0x5e005e5e,    0x47004747,    0x94009494,    0xfa00fafa,    0xfc00fcfc,
0x5b005b5b,
        0x97009797,    0xfe00fefe,    0x5a005a5a,    0xac00acac,    0x3c003c3c,
0x4c004c4c,
        0x03000303,    0x35003535,    0xf300f3f3,    0x23002323,    0xb800b8b8,
0x5d005d5d,
        0x6a006a6a,    0x92009292,    0xd500d5d5,    0x21002121,    0x44004444,
0x51005151,
        0xc600c6c6,    0x7d007d7d,    0x39003939,    0x83008383,    0xdc00dcdc,
0xaa00aaaa,
        0x7c007c7c,    0x77007777,    0x56005656,    0x05000505,    0x1b001b1b,
0xa400a4a4,
        0x15001515,    0x34003434,    0x1e001e1e,    0x1c001c1c,    0xf800f8f8,
0x52005252,
        0x20002020,    0x14001414,    0xe900e9e9,    0xbd00bdbd,    0xdd00dddd,
0xe400e4e4,
        0xa100a1a1,    0xe000e0e0,    0x8a008a8a,    0xf100f1f1,    0xd600d6d6,
0x7a007a7a,
        0xbb00bbbb, 0xe300e3e3, 0x40004040, 0x4f004f4f}
};

/* Key generation constants */
static const u32 SIGMA[] = {
    0xa09e667f,    0x3bcc908b,    0xb67ae858,    0x4caa73b2,    0xc6ef372f,
0xe94f82be,
    0x54ff53a5,    0xf1d36f1c,    0x10e527fa,    0xde682d1d,    0xb05688c2,
0xb3e6c1fd
};

/* The phi algorithm given in C.2.7 of the Camellia spec document. */
/*
 * This version does not attempt to minimize amount of temporary
 * variables, but instead explicitly exposes algorithm's parallelism.
 * It is therefore most appropriate for platforms with not less than
 * ~16 registers. For platforms with less registers [well, x86 to be
 * specific] assembler version should be/is provided anyway...
 */
#define Camellia_Feistel(_s0,_s1,_s2,_s3,_key) do {\
    u32 _t0,_t1,_t2,_t3;\
    _t0  = _s0 ^ (_key)[0];\
    _t3  = SBOX4_4404[_t0&0xff];\
    _t1  = _s1 ^ (_key)[1];\
    _t3 ^= SBOX3_3033[( _t0 >> 8)&0xff];\
    _t2  = SBOX1_1110[_t1&0xff];\
    _t3 ^= SBOX2_0222[( _t0 >> 16)&0xff];\

```

```

        _t2 ^= SBOX4_4404[(_t1 >> 8)&0xff];\
        _t3 ^= SBOX1_1110[(_t0 >> 24)];\
        _t2 ^= _t3;\
        _t3 = RightRotate(_t3,8);\
        _t2 ^= SBOX3_3033[(_t1 >> 16)&0xff];\
        _s3 ^= _t3;\
        _t2 ^= SBOX2_0222[(_t1 >> 24)];\
        _s2 ^= _t2; \
        _s3 ^= _t2;\
    } while(0)

#define Camellia_FeistelFile(_s0,_s1,_s2,_s3,_key,file) do {\
    u32 _t0,_t1,_t2,_t3;\
\
    _t0 = _s0 ^ (_key)[0];\
    _t3 = SBOX4_4404[_t0&0xff];\
    _t1 = _s1 ^ (_key)[1];\
    fwrite(&_t0, 4, 1, file);\
    fwrite(&_t1, 4, 1, file);\
    fwrite(&_s2, 4, 1, file);\
    fwrite(&_s3, 4, 1, file);\
    _t3 ^= SBOX3_3033[(_t0 >> 8)&0xff];\
    _t2 = SBOX1_1110[_t1&0xff];\
    _t3 ^= SBOX2_0222[(_t0 >> 16)&0xff];\
    _t2 ^= SBOX4_4404[(_t1 >> 8)&0xff];\
    _t3 ^= SBOX1_1110[(_t0 >> 24)];\
    _t2 ^= _t3;\
    _t3 = RightRotate(_t3,8);\
    _t2 ^= SBOX3_3033[(_t1 >> 16)&0xff];\
    _s3 ^= _t3;\
    _t2 ^= SBOX2_0222[(_t1 >> 24)];\
    _s2 ^= _t2; \
    _s3 ^= _t2;\
} while(0)

/*
 * Note that n has to be less than 32. Rotations for larger amount
 * of bits are achieved by "rotating" order of s-elements and
 * adjusting n accordingly, e.g. RotLeft128(s1,s2,s3,s0,n-32).
 */
#define RotLeft128(_s0,_s1,_s2,_s3,_n) do {\
    u32 _t0=_s0>>(32-_n);\
    _s0 = (_s0<<_n) | (_s1>>(32-_n));\
    _s1 = (_s1<<_n) | (_s2>>(32-_n));\
    _s2 = (_s2<<_n) | (_s3>>(32-_n));\
    _s3 = (_s3<<_n) | _t0;\
} while (0)

int Camellia_Ekeygen(int keyBitLength, const u8 *rawKey, KEY_TABLE_TYPE
k)
{
    register u32 s0, s1, s2, s3;

    k[0] = s0 = GETU32(rawKey);
    k[1] = s1 = GETU32(rawKey + 4);

```

```

k[2] = s2 = GETU32(rawKey + 8);
k[3] = s3 = GETU32(rawKey + 12);

if (keyBitLength != 128) {
    k[8] = s0 = GETU32(rawKey + 16);
    k[9] = s1 = GETU32(rawKey + 20);
    if (keyBitLength == 192) {
        k[10] = s2 = ~s0;
        k[11] = s3 = ~s1;
    } else {
        k[10] = s2 = GETU32(rawKey + 24);
        k[11] = s3 = GETU32(rawKey + 28);
    }
    s0 ^= k[0], s1 ^= k[1], s2 ^= k[2], s3 ^= k[3];
}

/* Use the Feistel routine to scramble the key material */
Camellia_Feistel(s0, s1, s2, s3, SIGMA + 0);
Camellia_Feistel(s2, s3, s0, s1, SIGMA + 2);

s0 ^= k[0], s1 ^= k[1], s2 ^= k[2], s3 ^= k[3];
Camellia_Feistel(s0, s1, s2, s3, SIGMA + 4);
Camellia_Feistel(s2, s3, s0, s1, SIGMA + 6);

/* Fill the keyTable. Requires many block rotations. */
if (keyBitLength == 128) {
    k[4] = s0, k[5] = s1, k[6] = s2, k[7] = s3;
    RotLeft128(s0, s1, s2, s3, 15); /* KA <<< 15 */
    k[12] = s0, k[13] = s1, k[14] = s2, k[15] = s3;
    RotLeft128(s0, s1, s2, s3, 15); /* KA <<< 30 */
    k[16] = s0, k[17] = s1, k[18] = s2, k[19] = s3;
    RotLeft128(s0, s1, s2, s3, 15); /* KA <<< 45 */
    k[24] = s0, k[25] = s1;
    RotLeft128(s0, s1, s2, s3, 15); /* KA <<< 60 */
    k[28] = s0, k[29] = s1, k[30] = s2, k[31] = s3;
    RotLeft128(s1, s2, s3, s0, 2); /* KA <<< 94 */
    k[40] = s1, k[41] = s2, k[42] = s3, k[43] = s0;
    RotLeft128(s1, s2, s3, s0, 17); /* KA <<<111 */
    k[48] = s1, k[49] = s2, k[50] = s3, k[51] = s0;

    s0 = k[0], s1 = k[1], s2 = k[2], s3 = k[3];
    RotLeft128(s0, s1, s2, s3, 15); /* KL <<< 15 */
    k[8] = s0, k[9] = s1, k[10] = s2, k[11] = s3;
    RotLeft128(s0, s1, s2, s3, 30); /* KL <<< 45 */
    k[20] = s0, k[21] = s1, k[22] = s2, k[23] = s3;
    RotLeft128(s0, s1, s2, s3, 15); /* KL <<< 60 */
    k[26] = s2, k[27] = s3;
    RotLeft128(s0, s1, s2, s3, 17); /* KL <<< 77 */
    k[32] = s0, k[33] = s1, k[34] = s2, k[35] = s3;
    RotLeft128(s0, s1, s2, s3, 17); /* KL <<< 94 */
    k[36] = s0, k[37] = s1, k[38] = s2, k[39] = s3;
    RotLeft128(s0, s1, s2, s3, 17); /* KL <<<111 */
    k[44] = s0, k[45] = s1, k[46] = s2, k[47] = s3;

    return 3; /* grand rounds */
}

```

```

} else {
    k[12] = s0, k[13] = s1, k[14] = s2, k[15] = s3;
    s0 ^= k[8], s1 ^= k[9], s2 ^= k[10], s3 ^= k[11];
    Camellia_Feistel(s0, s1, s2, s3, (SIGMA + 8));
    Camellia_Feistel(s2, s3, s0, s1, (SIGMA + 10));

    k[4] = s0, k[5] = s1, k[6] = s2, k[7] = s3;
    RotLeft128(s0, s1, s2, s3, 30); /* KB <<< 30 */
    k[20] = s0, k[21] = s1, k[22] = s2, k[23] = s3;
    RotLeft128(s0, s1, s2, s3, 30); /* KB <<< 60 */
    k[40] = s0, k[41] = s1, k[42] = s2, k[43] = s3;
    RotLeft128(s1, s2, s3, s0, 19); /* KB <<<111 */
    k[64] = s1, k[65] = s2, k[66] = s3, k[67] = s0;

    s0 = k[8], s1 = k[9], s2 = k[10], s3 = k[11];
    RotLeft128(s0, s1, s2, s3, 15); /* KR <<< 15 */
    k[8] = s0, k[9] = s1, k[10] = s2, k[11] = s3;
    RotLeft128(s0, s1, s2, s3, 15); /* KR <<< 30 */
    k[16] = s0, k[17] = s1, k[18] = s2, k[19] = s3;
    RotLeft128(s0, s1, s2, s3, 30); /* KR <<< 60 */
    k[36] = s0, k[37] = s1, k[38] = s2, k[39] = s3;
    RotLeft128(s1, s2, s3, s0, 2); /* KR <<< 94 */
    k[52] = s1, k[53] = s2, k[54] = s3, k[55] = s0;

    s0 = k[12], s1 = k[13], s2 = k[14], s3 = k[15];
    RotLeft128(s0, s1, s2, s3, 15); /* KA <<< 15 */
    k[12] = s0, k[13] = s1, k[14] = s2, k[15] = s3;
    RotLeft128(s0, s1, s2, s3, 30); /* KA <<< 45 */
    k[28] = s0, k[29] = s1, k[30] = s2, k[31] = s3;
    /* KA <<< 77 */
    k[48] = s1, k[49] = s2, k[50] = s3, k[51] = s0;
    RotLeft128(s1, s2, s3, s0, 17); /* KA <<< 94 */
    k[56] = s1, k[57] = s2, k[58] = s3, k[59] = s0;

    s0 = k[0], s1 = k[1], s2 = k[2], s3 = k[3];
    RotLeft128(s1, s2, s3, s0, 13); /* KL <<< 45 */
    k[24] = s1, k[25] = s2, k[26] = s3, k[27] = s0;
    RotLeft128(s1, s2, s3, s0, 15); /* KL <<< 60 */
    k[32] = s1, k[33] = s2, k[34] = s3, k[35] = s0;
    RotLeft128(s1, s2, s3, s0, 17); /* KL <<< 77 */
    k[44] = s1, k[45] = s2, k[46] = s3, k[47] = s0;
    RotLeft128(s2, s3, s0, s1, 2); /* KL <<<111 */
    k[60] = s2, k[61] = s3, k[62] = s0, k[63] = s1;

    return 4; /* grand rounds */
}
/*
 * It is possible to perform certain precalculations, which
 * would spare few cycles in block procedure. It's not done,
 * because it upsets the performance balance between key
 * setup and block procedures, negatively affecting overall
 * throughput in applications operating on short messages
 * and volatile keys.
 */
}

```



```

void Camellia_EncryptBlock_Rounds(int grandRounds, const u8 plaintext[],
                                   const KEY_TABLE_TYPE keyTable,
                                   u8 ciphertext[], FILE** files)
{
    u32 s0, s1, s2, s3;
    const u32 *k = keyTable, *kend = keyTable + grandRounds * 16;

    s0 = GETU32(plaintext) ^ k[0];
    s1 = GETU32(plaintext + 4) ^ k[1];
    s2 = GETU32(plaintext + 8) ^ k[2];
    s3 = GETU32(plaintext + 12) ^ k[3];
    k += 4;

    fwrite(&s0, 4, 1, files[0]);
    fwrite(&s1, 4, 1, files[0]);
    fwrite(&s2, 4, 1, files[0]);
    fwrite(&s3, 4, 1, files[0]);

    int roundGo = 0;

    while (1) {
        /* Camellia makes 6 Feistel rounds */
        Camellia_FeistelFile(s0, s1, s2, s3, k + 0, files[roundGo + 1]);
        Camellia_FeistelFile(s2, s3, s0, s1, k + 2, files[roundGo + 2]);
        Camellia_FeistelFile(s0, s1, s2, s3, k + 4, files[roundGo + 3]);
        Camellia_FeistelFile(s2, s3, s0, s1, k + 6, files[roundGo + 4]);
        Camellia_FeistelFile(s0, s1, s2, s3, k + 8, files[roundGo + 5]);
        Camellia_FeistelFile(s2, s3, s0, s1, k + 10, files[roundGo + 6]);
        k += 12;
        roundGo += 6;

        if (k == kend)
            break;

        /*
         * This is the same function as the diffusion function D of the
         * accompanying documentation. See section 3.2 for properties of
the
         * FLlayer function.
         */
        s1 ^= LeftRotate(s0 & k[0], 1);
        s2 ^= s3 | k[3];
        s0 ^= s1 | k[1];
        s3 ^= LeftRotate(s2 & k[2], 1);
        ++roundGo;
        k += 4;

        fwrite(&s0, 4, 1, files[roundGo]);
        fwrite(&s1, 4, 1, files[roundGo]);
        fwrite(&s2, 4, 1, files[roundGo]);
        fwrite(&s3, 4, 1, files[roundGo]);
    }

    s2 ^= k[0], s3 ^= k[1], s0 ^= k[2], s1 ^= k[3];

```

```

        ++roundGo;
        fwrite(&s2, 4, 1, files[roundGo]);
        fwrite(&s3, 4, 1, files[roundGo]);
        fwrite(&s0, 4, 1, files[roundGo]);
        fwrite(&s1, 4, 1, files[roundGo]);

        PUTU32(ciphertext, s2);
        PUTU32(ciphertext + 4, s3);
        PUTU32(ciphertext + 8, s0);
        PUTU32(ciphertext + 12, s1);
    }

void Camellia_EncryptBlock(int keyBitLength, const u8 plaintext[],
                           const KEY_TABLE_TYPE keyTable, u8
ciphertext[], FILE** files)
{
    Camellia_EncryptBlock_Rounds(keyBitLength == 128 ? 3 : 4,
                                  plaintext, keyTable, ciphertext,
files);
}

```

camellia.h

```

#ifndef OSSL_CRYPTO_CAMELLIA_CMLL_LOCAL_H
# define OSSL_CRYPTO_CAMELLIA_CMLL_LOCAL_H

#include <stdio.h>

# define CAMELLIA_TABLE_BYTE_LEN 272
# define CAMELLIA_TABLE_WORD_LEN (CAMELLIA_TABLE_BYTE_LEN / 4)

typedef unsigned int KEY_TABLE_TYPE[CAMELLIA_TABLE_WORD_LEN];

typedef unsigned int u32;
typedef unsigned char u8;

int Camellia_Ekeygen(int keyBitLength, const u8 *rawKey,
                     KEY_TABLE_TYPE keyTable);
void Camellia_EncryptBlock_Rounds(int grandRounds, const u8 plaintext[],
                                   const KEY_TABLE_TYPE keyTable,
                                   u8 ciphertext[], FILE** files);
void Camellia_EncryptBlock(int keyBitLength, const u8 plaintext[],
                           const KEY_TABLE_TYPE keyTable, u8
ciphertext[], FILE** files);

#endif

```

run_tests.sh – автоматизация nist_sts

```

#!/usr/bin/env bash

INPUT_DIR="./input_data"
OUTPUT_DIR="./nist_results"

```

```

PARAM_ANSWERS="./answers.txt"
ASSESS="./assess"
EXPERIMENT_DIR="./experiments"
REPORT_PATH="$EXPERIMENT_DIR/AlgorithmTesting/finalAnalysisReport.txt"
BITS=1000000

mkdir -p "$OUTPUT_DIR"

for file in "$INPUT_DIR"/round_*.bin; do
    [ -f "$file" ] || continue

    name=$(basename "$file")
    prefix="{name%.bin}"
    num_part="{prefix#round_}"
    num=$((10#$num_part))

    output_file="$OUTPUT_DIR/$num.txt"

    echo "Обработка файла: $file -> $output_file"

    # Копируем текущий входной файл в input.bin
    cp "$file" input.bin

    # Запускаем assess с передачей длины битстрима в аргументах и ответов
    из answers.txt
    $ASSESS $BITS < "$PARAM_ANSWERS"

    if [ -f "$REPORT_PATH" ]; then
        cp "$REPORT_PATH" "$output_file"
        echo "Результат сохранен в $output_file"
    else
        echo "Отчет не найден: $REPORT_PATH. Проверьте корректность
запуска."
    fi

    echo "-----"
done

echo "Все тесты завершены."

```

read_binary.c

```

#include <stdio.h>
#include <stdlib.h>
#include "unif01.h"
#include "bbattery.h"

// Глобальные переменные для хранения данных
unsigned int *buffer = NULL;
size_t buffer_size = 0;
size_t current_index = 0;

// Функция чтения данных из буфера
unsigned int ReadBinary(void) {
    if (current_index >= buffer_size) {

```

```

        return 0; // Конец данных
    }
    return buffer[current_index++];
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <binary file>\n", argv[0]);
        return 1;
    }

    // Открываем бинарный файл
    FILE *file = fopen(argv[1], "rb");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    // Определяем размер файла
    fseek(file, 0, SEEK_END);
    size_t file_size = ftell(file);
    rewind(file);

    // Проверяем, что файл делится на 4 байта
    if (file_size % sizeof(unsigned int) != 0) {
        printf("File size is not a multiple of 4 bytes.\n");
        fclose(file);
        return 1;
    }

    // Читаем весь файл в память
    buffer_size = file_size / sizeof(unsigned int);
    buffer = (unsigned int *)malloc(buffer_size * sizeof(unsigned int));
    if (!buffer) {
        perror("Memory allocation failed");
        fclose(file);
        return 1;
    }

    if (fread(buffer, sizeof(unsigned int), buffer_size, file) !=
buffer_size) {
        perror("Error reading file");
        free(buffer);
        fclose(file);
        return 1;
    }
    fclose(file);

    // Создаем генератор
    unif01_Gen *gen = unif01_CreateExternGenBits("Binary File Reader",
ReadBinary);
    // Запускаем статистические тесты
    bbattery_SmallCrush(gen);
    // Удаляем генератор и освобождаем память
    unif01_DeleteExternGenBits(gen);

```

```
    free(buffer);  
  
    return 0;  
}
```