МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования   
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 02 Информационные системы и технологии

Направление специальности 1-40 01 02 03 Информационные системы и технологии (издательско-полиграфический комплекс)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**КУРСОВОГО ПРОЕКТА:**

по дисциплине «Защита информации и надежность информационных систем»

Тема: «Реализация и исследования алгоритма хеширования STREEBOG»

Исполнитель

Студент 3 курса группы 3 Шедько Е.А.

(Ф.И.О.)

Руководитель работы ассистент Копыток Д.В. (учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Председатель Берников В.О.

(подпись, дата)

Минск 2022

**Содержание**

[Введение 3](#_Toc103000493)

[1. Постановка задачи 4](#_Toc103000494)

[2. Аналитический обзор литературы 5](#_Toc103000495)

[3. Криптостойкость алгоритма 11](#_Toc103000496)

[4. Сравнение ГОСТ 34.11-2012 и ГОСТ 34.11-94 13](#_Toc103000497)

[5. Описание программного средства 15](#_Toc103000498)

[6. Тестирование программного средства 20](#_Toc103000499)

[7. Руководство пользователя 25](#_Toc103000500)

[Заключение 29](#_Toc103000501)

[Список используемых источников 30](#_Toc103000502)

[ПРИЛОЖЕНИЕ А 31](#_Toc103000503)

# **Введение**

Криптографические хеш-функции — незаменимый и повсеместно распространенный инструмент, используемый для выполнения целого ряда задач, включая аутентификацию, проверку целостности данных, защиту файлов и даже обнаружение зловредного ПО, в криптовалютных технологиях. Хеш-функция представляет собой математическую функцию, которая принимает на входе строку символов, называемую также прообразом, переменной длины и преобразует ее в выходную строку фиксированной длины. Преобразование, производимое хеш-функцией, называется хешированием, а результат преобразования хешем. Вычисляемый хеш всегда уникален, для каждого массива информации. Однако иногда случаются так называемые коллизии, когда для разных входных блоков информации вычисляются одинаковые хеш-коды. Хеш-функции разрабатываются таки образом, что даже при самом незначительном изменении входной информации ее хеш полностью изменяется. Также важной особенностью хеш-функций является то, что она необратима и не позволяет восстанавливать исходный массив информации из символьной строки(хеша). Это возможно осуществить, только перебрав все возможные варианты, что при бесконечном количестве информации требует больших вычислительных ресурсов и большое количество времени. Данная курсовая работа посвящена изучению хеш-функции под названием STREEBOG.

STREEBOG – криптографический алгоритм вычисления хеш-функции с размером блока входных данных 512 бит и размером хеш-кода 256 или 512 бит. Описывается в ГОСТ 34.11-2018 «Информационная технология. Криптографическая защита информации» - действующем межгосударственном криптографическом стандарте. Разработан Центром защиты информации и специальной связи ФСБ России с участием ОАО «ИнфоТеКС» на основе национального стандарта Российской Федерации ГОСТ Р 34.11-2012 и введен в действие с 1 июня 2019 года.

# **Постановка задачи**

Требуется разработать приложение, которое демонстрирует работу алгоритма хеширования STREEBOG и его составных частей.

Данными, для работы алгоритма будет служить текстовая информация.

Требования к средствам разработки:

* язык программирования C#.
* средство разработки MS Visual Studio.

Программа должна:

* осуществлять вычисление хеша входных данных при помощи алгоритма хеширования STREEBOG;
* предоставлять возможность ввода данных, необходимых для вычисления хеша, путем заполнения текстового блока;
* предоставлять возможность ввода данных, необходимых для вычисления хеша, путем взятия их из текстового файла;
* предоставлять выбор конечного размера хеш-кода(512 или 256 бит).
* отображать результат вычисления хеша сообщения;
* осуществлять запись результата вычисления хеша в текстовый файл.
* отображать результаты работы промежуточных частей работы алгоритма хеширования.

Программа также должна иметь простой, удобный и понятный графический интерфейс.

# **Аналитический обзор литературы**

В основу хеш-функции STREEBOG положена итерационная конструкция Меркла – Дамгора[1] с использованием MD-усиления. Конструкция Меркла – Дамгора – это метод построения криптографических хеш-функция, предусматривающий разбиение входных сообщений произвольной длины на блоки фиксированной длины и работающий с ними по очереди с помощью функции сжатия, каждый раз принимая входной блок с выходным от предыдущего прохода (рисунок 2.1).

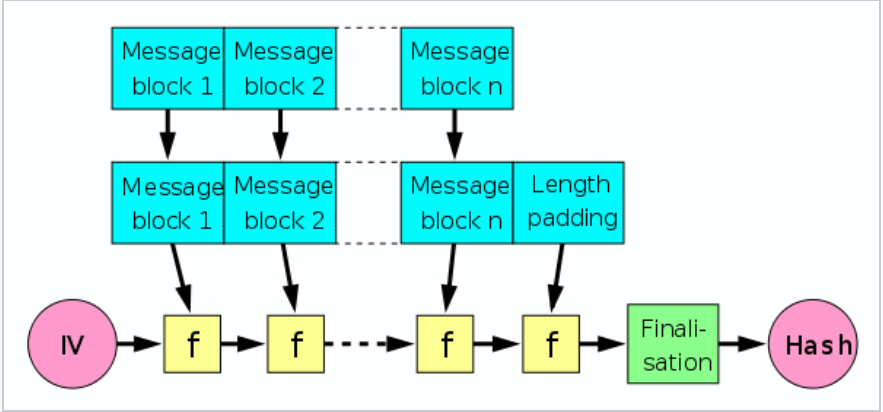


Рисунок 2.1 – Итерационная конструкция Меркла – Дамгора

Структура предусматривает вектор инициализации – фиксированное значение, которое зависит от реализации алгоритма. Под MD-усилением понимается дополнение неполного блока при вычислении хеш-функции до полного путем добавления вектора (0…01) такой длины, чтобы получился полный блок. Из дополнительных элементов нужно отметить следующие:

* завершающее преобразование, которое заключается в том, что функция сжатия применяется к контрольной сумме всех блоков сообщения по модулю 2^512.
* при вычислении хеш-кода на каждой итерации применяются разные функции сжатия. Можно сказать, что функция сжатия зависит от номера итерации.

Общая схема работы алгоритма хеширования STREEBOG, основанная на конструкции Меркла – Дамгора представлена на рисунке 2.2

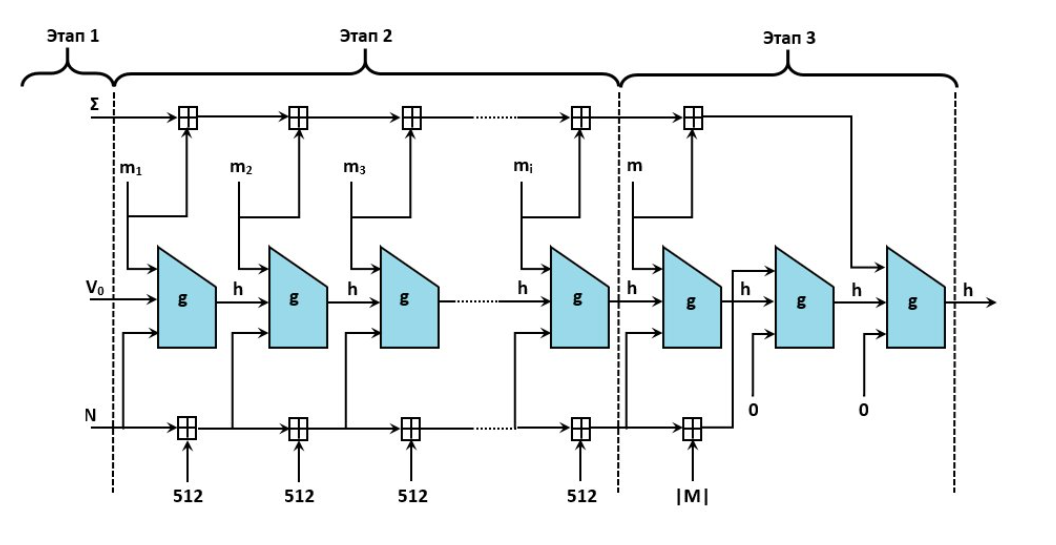


Рисунок 2.2 – Общая схема работы алгоритма хеширования STREEBOG

Хеширования производится в три этапа. Первый этап – инициализация всех нужных параметров, второй этап представляет собой конструкцию Маркла-Дамгора с процедурой MD-усиления, третий этап – завершающее преобразование: функция сжатия применяется к сумме всех блоков сообщения и дополнительно хешируется длина сообщения и его контрольная сумма. Хеш-функция способна генерировать хеш длинной 512 и 256 бит. В зависимости от размера хеша применяется различные значения инициализирующего вектора (рисунок 2.3)



Рисунок 2.3 – Значения инициализирующего вектора

Как видно из рисунка 2.3 значение инициализирующего вектора, для хеш-функции с размером выходного хеша в 512, бит составляет 0, в то время, как при значении выходного хеша в 256 бит, каждый байт инициализирующего вектора равен единице. На вход хеш-функции подается сообщение произвольного размера. Далее сообщение разбивается на блоки по 512 бит, если размер сообщения не кратен 512, то оно дополняется последовательностью бит необходимой длины, которая состоит из нулей и последнего бита, равного единице. Далее итерационно используется функция сжатия. В рассматриваемом алгоритме хеширования функция сжатия построена на конструкции Миагучи – Пренеля [2] (рисунок 2.4).

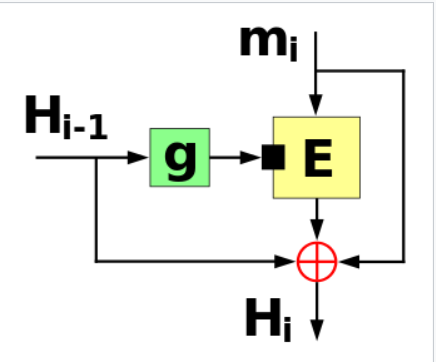


Рисунок 2.4 – Конструкция Миагучи-Пренеля

Конструкция предусматривает подачу каждого блока исходного сообщения в качестве текста, который будет зашифрован. Далее выполняется операция XOR зашифрованного текста с тем же блоком исходного сообщения, а затем также операции XOR c предыдущим значением хеш-функции, для получения следующего значения хеша. Предыдущее значение хеш-функции подается в качестве ключа для блочного шифра. В первом раунде, когда нет предыдущего значения хеша, он использует заранее постоянное оговоренное начальной значение. В хеш-функции STREEBOOG это значение является 512-битной последовательностью, состоящей из нулей. Схема работы функции сжатия представлена на рисунке 2.5

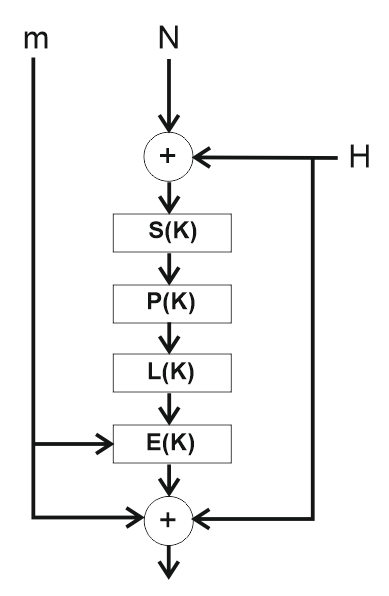


Рисунок 2.5 - Схема работы функции сжатия алгоритма хеширования STREEBOG

Функция E(k, m) – блочный шифр с длиной блока и ключа 512 бит, схема которого представлена на рисунке 2.6.

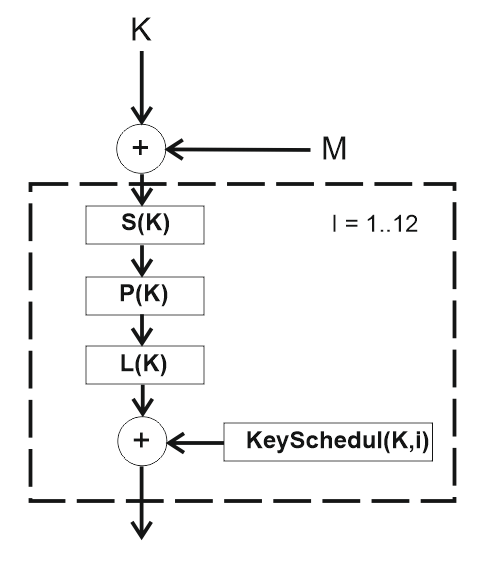


Рисунок 2.6 – Схема работы блочного шифра алгоритма хеширования STREEBOG

В качестве блочного шифра в хеш-функции взять XSPL – шифр. Данный шифр состоит из 4 основных операций:

* сложение по модулю 2
* нелинейное биективное преобразование множества двоичных векторов (S);
* перестановка байт (P);
* линейное преобразование множества двоичных векторов (L);

Нелинейное биективное преобразование множества двоичных векторов задается подстановкой. Значения подстановки задаются в виде массива, представленного на рисунке 2.7

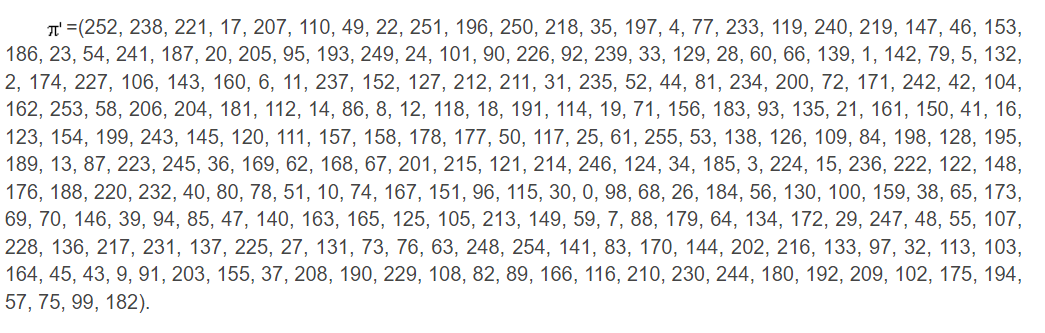


Рисунок 2.7 – Массив подстановок нелинейного биективного преобразования

Операция перестановки. Значения перестановок записаны в виде массива, представленного на рисунке 2.8

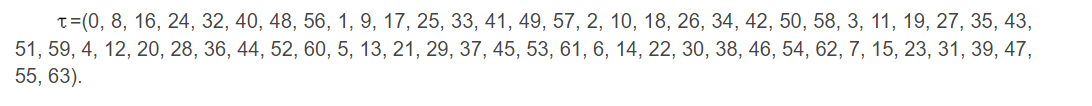


Рисунок 2.8 – Массив перестановок

Линейное преобразование множества двоичных векторов задается умножением справа на матрицу А над полем GF (2), строки которой записаны последовательно в шестнадцатеричном виде. Исходный вектор делится на порции размером по 8 байт, каждая из этих порций интерпретируется в виде 64-разрядного двоичного представления. Далее берется очередная порция, каждому биту из этой порции ставится в соответствие строка из матрицы A. Если очередной бит равен нулю, то соответствующая ему строка из матрицы A вычеркивается; если очередной бит равен единице, то соответствующая ему строка из матрицы A остается. После всего этого оставшиеся строки из матрицы линейного преобразования слаживаются операцией XOR, и получившееся значение записывается в виде очередной восьмибайтовой порции в результирующий вектор. Значения матрицы А представлены на рисунке 2.9.

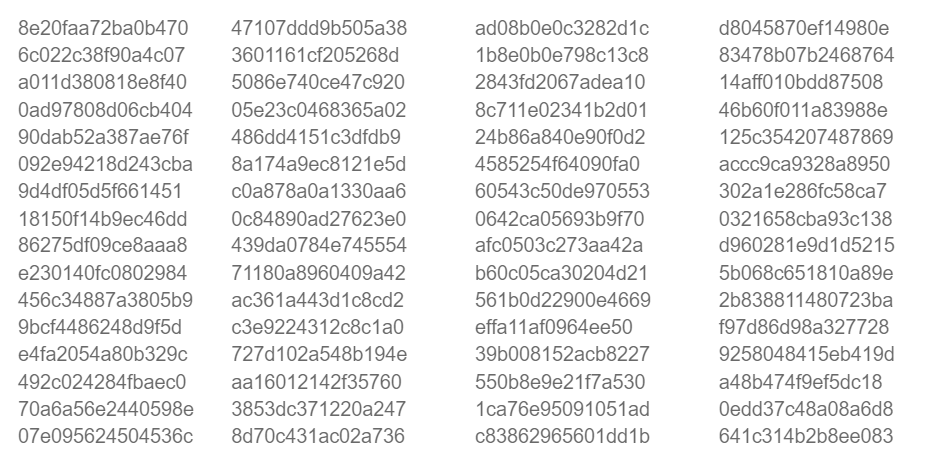


Рисунок 2.9 – Матрица А линейного преобразования множества двоичных векторов

Важным параметром блочного шифра является то, как выбирается ключ, который будет использоваться на каждом раунде преобразования. В применяемом блочном шифре ключи, для каждого из 13 раундов, генерируются с помощью самой функции шифрования. Схема генерации ключей представлена на рисунке 2.10

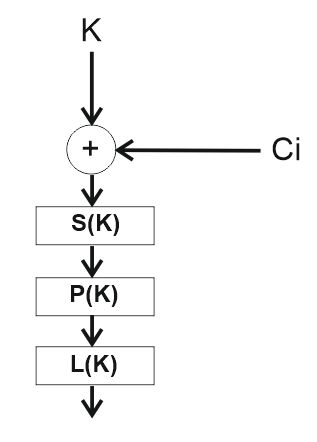


Рисунок 2.10 – Схема формирования ключа, для раундов блочного алгоритма шифрования

Сi – итерационные константы, которые являются битовыми векторами. Их значения указаны в соответствующем разделе стандарта [3].

# **Криптостойкость алгоритма**

Исходя из конструкции новой хеш-функции, можно сделать некоторые выводы о её криптостойкости и предположить, что её криптостойкость будет выше, чем у ГОСТ Р 34.11-94(предыдущий стандарт):

* все блоки сообщения суммируются по модулю 2512 и уже результат

суммирования всех блоков подается на вход завершающего этапа (stage3).

Благодаря тому, что здесь суммирование — это не побитовое сложение, получается защита от следующих атак:

* + - построение мультиколлизий;
    - удлинение прообраза;
    - дифференциальный криптоанализ
* в функции сжатия используется конструкция Миагути — Пренели, это обеспечивает защиту от атаки, основанную на фиксированных точках, так как для конструкции Миагути — Пренели не найдено лёгких способов для поиска фиксированных точек.
* на каждой итерации при вычислении хеш-кода используются различные константы. Это затрудняет атаки на основе связанных и разностных связанных ключей, атаки скольжения и отражения.

В 2013 году на сайте «Cryptology ePrint Archive: Listing for 2013» было опубликовано две статьи, посвящённых криптоанализу новой хеш-функции. В статье «Rebound attack on Stribog»исследуется устойчивость хеш-функции по отношению к атаке, называемой «The Rebound attack»; в основе этой атаки лежит «rotation cryptanalysis» и дифференциальный криптоанализ. Криптоаналитики при поиске уязвимостей использовали метод, называемый «free-start». Это означает, что при вычислении хеш-кода фиксируется некоторое состояние хеш-функции и дальше вычисления могут идти как в сторону вычисления хеш-кода, так и в сторону вычисления сообщения. Криптоаналитики сумели добиться коллизии за 5 раундов и была получена так называемая «near collision» (это означает, что были найдены два сообщения, хеш-коды которых отличны в малом количестве бит) при использовании 7,75 раундов. Также было установлено, что схема, по которой выбираются ключи для каждого раунда, добавляет устойчивости функции сжатия. Однако было показано, что коллизия возможна за 7,75 раундов, а «near collision» — за 8,75 и 9,75, соответственно.

В статье «Integral Distinguishers for Reduced-round Stribog»рассматривается стойкость хеш-функции (с уменьшенным количеством раундов) по отношению к интегральному криптоанализу. Авторами при исследовании функции сжатия удалось найти дифференциал за 4 раунда при вычислении в прямом направлении и за 3,5 раунда при вычислении в обратном направлении. Также было выяснено, что атака нахождения дифференциала на хеш-функцию с числом раундов 6 и 7 требует 264 и 2120 среднераундовых значений, соответственно.

Для изучения криптостойкости новой хеш-функции компания «ИнфоТеКС» в ноябре 2013 года объявила о старте конкурса; он завершился в мае 2015 года. Победителем стала работа «The Usage of Counter Revisited: Second-Preimage Attack on New Russian Standardized Hash Function», в которой авторы представили атаку нахождения второго прообраза для хеш-функции «Стрибог-512», требующую 2266 вызовов функции сжатия для сообщений длиннее 2259 блоков.

# **Сравнение ГОСТ 34.11-2012 и ГОСТ 34.11-94**

Основные отличия ГОСТ 34.11-2012 и ГОСТ 34.11-94:

* в ГОСТ Р 34.11-2012 размер блоков сообщения и внутреннего состояния хеш-функции составляет 512 бит против 256 бит в ГОСТ Р 34.11-94.
* новый стандарт определяет две функции хеширования с длинами хеш-кода 256 и 512 бит, в то время как в старом стандарте длина хеш-кода может быть только 256 бит. Возможность вариации выходного хеша может быть полезна в случае встроенных реализаций с ограниченными ресурсами или наличия каких-то дополнительных требований в области криптографии.
* основное отличие современной хеш-функции от старой — функция сжатия. В ГОСТ Р 34.11-2012 используется функции сжатия, в основе которой лежат три преобразования: нелинейное биективное преобразование (обозначается S), перестановка байт (обозначается P), линейное преобразование(обозначается L). В ГОСТ Р 34.11-94 используется функция сжатия, основанная на симметричном блочном шифре ГОСТ Р 28147-89, также эта функция использует операции перемешивания.
* при вычислении новой хеш-функции, если размер сообщения не кратен размеру обрабатываемого блока (для современного стандарта — 512 бит, для старого стандарта — 256 бит), то такой блок дополняется вектором (00 … 01). При вычислении старой хеш-функции неполный блок дополняется значением (00 … 0). Считается, что дополнение (00 … 01) лучше, чем (00 … 0), с криптографической точки зрения, так как дополнения значением (00 … 0) приводит к атакам Оракула дополнения. В случае ненулевого дополнения была доказана стойкость к подобным атакам.
* ещё одно отличие состоит в том, что стандарт ГОСТ Р 34.11-94 не определял значение инициализационного вектора, в то время как в стандарте ГОСТ Р 34.11-2012 значение инициализационного вектора фиксировано и определено в стандарте: для хеш-функции с размером выходного хеша 512 бит это вектор (00 … 0), для хеш-функции с размером выходного хеш-кода 256 бит — (000000010 … 100000001) (все байты равны 1).

Также имеются различия в быстродействии алгоритмов. Для сравнения быстродействия на процессоре архитектуры x86\_64 были взяты 4 разных реализации хеш-функций:

1. реализация ГОСТ Р 34.11-1994 из криптографического пакета OpenSSL (версия 1.0.1c) с открытым исходным кодом. В этой реализации нет алгоритмических и программных оптимизаций;
2. реализация ГОСТ Р 34.11-1994 в программе RHash (версия 1.2.9). В этой реализации есть алгоритмические и программные оптимизации, в том числе ассемблерные оптимизации;
3. реализация ГОСТ Р 34.11-2012, написанная А. Казимировым;
4. реализации ГОСТ Р 34.11-1994 и ГОСТ Р 34.11-2012, написанные П.А. Лебедевым.

Использовался процессор Intel Core i7-920 CPU на базовой частоте 2,67 ГГц. Результаты производительности(рисунок 4.1):

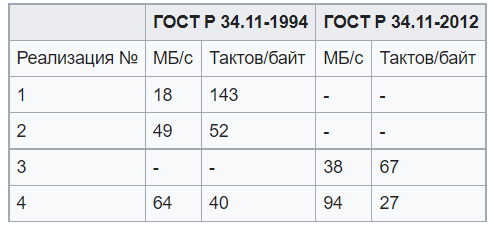


Рисунок 4.1 – Результаты производительности ГОСТ 34.11-1994 и ГОСТ Р 34.11-2012(процессор Intel Core i7-920 CPU )

Сравнение быстродействия старого и нового стандартов хеш-функций на GPU проводилось между реализациями П.А. Лебедева. Использовалась видеокарта NVIDIA GTX 580. Результаты производительности (рисунок 4.2):

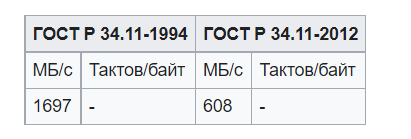


Рисунок 4.2 - Результаты производительности ГОСТ 34.11-1994 и ГОСТ Р 34.11-2012(NVIDIA GTX 580)

На основании этих результатов можно сделать вывод, что хеш-функция ГОСТ Р 34.11-2012 в два раза быстрее хеш-функции ГОСТ Р 34.11-94 на современных процессорах, но медленнее на графических картах и системах с ограниченными ресурсами.

Такие результаты производительности можно объяснить тем, что при вычислении новой хеш-функции используются только сложения по модулю 2 и инструкции пересылки данных. Старая хеш-функциия содержит много инструкций перемешивания, которые не лучшим образом отображаются на набор команд ЦП. Но увеличенный размер состояний и таблиц подстановки хеш-функции ГОСТ Р 34.11-2012 делает её медленней на высокопараллельных вычислительных средствах, таких как графические процессоры.

Также исследование производительности новой хеш-функции было проведено её разработчиками на 64-битном процессоре Intel Xeon E5335 2 ГГц. Использовалось одно ядро. Производительность хеш-функции ГОСТ Р 34.11-2012 составила 51 такт процессора на 1 байт хешируемых данных (примерно 40 MБ/с). Полученный результат на 20% лучше, чем у старой хеш-функции ГОСТ Р 34.11-94.

# **Описание программного средства**

Для разработки программного средства была выбрана платформа .NET и язык C#.

.NET Framework – программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для различных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду. В настоящее время .NET Framework развивается в виде .NET.

C# – объектно-ориентированный язык программирования. Разработан в 1998–2001 годах группой инженеров компании Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота как язык разработки приложений для платформы Microsoft .NET Framework[4]. Впоследствии был стандартизирован как ECMA-334 и ISO/IEC 23270.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

В качестве среды для разработки была выбрана Visual Studio 2019[5]. Visual Studio – это интегрированная среда разработки (IDE) от компании Microsoft. С помощью нее можно разрабатывать:

* классические приложения для компьютера под управлением операционной системы Windows;
* мобильные приложения (Windows, iOS, Android);
* web-приложения;
* облачные приложения;
* различные расширения для Office, SharePoint, а также создание собственных расширений для Visual Studio;
* игры;
* базы данных SQL Server и SQL Azure.

Visual Studio это мощный инструмент, который позволяет использовать следующие технологии и языки программирования: .NET, Node.js, C, C#, C++, Python, Visual Basic, F#, JavaScript.

Windows Presentation Foundation (WPF) – аналог WinForms, система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая (презентационная) подсистема в составе .NET Framework (начиная с версии 3.0), использующая язык XAML.

Для обеспечения комфортной работы и расширяемости приложения, архитектурный паттерн MVVM. Model-View-ViewModel (MVVM) – шаблон проектирования архитектуры приложения. Представлен в 2005 году Джоном Госсманом (John Gossman) как модификация шаблона Presentation Model. Используется для разделения модели и её представления, что необходимо для их изменения отдельно друг от друга. Шаблон MVVM делится на три части:

* модель (англ. Model) (так же, как в классической MVC) представляет собой логику работы с данными и описание фундаментальных данных, необходимых для работы приложения.
* представление (англ. View) – графический интерфейс (окна, списки, кнопки и т. п.). Выступает подписчиком на событие изменения значений свойств или команд, предоставляемых Моделью Представления. В случае, если в Модели Представления изменилось какое-либо свойство, то она оповещает всех подписчиков об этом, и Представление, в свою очередь, запрашивает обновлённое значение свойства из Модели Представления. В случае, если пользователь воздействует на какой-либо элемент интерфейса, Представление вызывает соответствующую команду, предоставленную Моделью Представления.
* модель Представления (англ. ViewModel) – с одной стороны, абстракция Представления, а с другой – обёртка данных из Модели, подлежащих связыванию. То есть, она содержит Модель, преобразованную к Представлению, а также команды, которыми может пользоваться Представление, чтобы влиять на Модель.

На рисунке 4.1 представлена структура проекта STREEBOGInterface. Проект представляет собой WPF приложение, которое демонстрирует подробный принцип работы алгоритма хеширования STREEBOG.

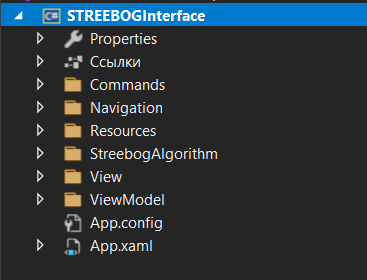


Рисунок 4.1 – Структура проекта STREEBOGInterface

Папка Commands содержит два класса BaseCommand.cs и DelegateCommand.cs. Данные классы предназначены для построения паттерна MVVM на основе механизма команд.

Папка Navigation содержит в себе классы и интерфейсы: INavigationActions.cs, INavigationManager.cs, NavigationKeys, NavigationManager, ParametersForWindow.cs. Основными в данной папке являются два класса: NavigationKeys.cs, который хранит ключи, на основе которых меняются окна в приложении; NavigationManager.cs, который собственно и осуществляет смену окон в приложении. Структура класса NavigationManager.cs представлена на рисунке 4.2.

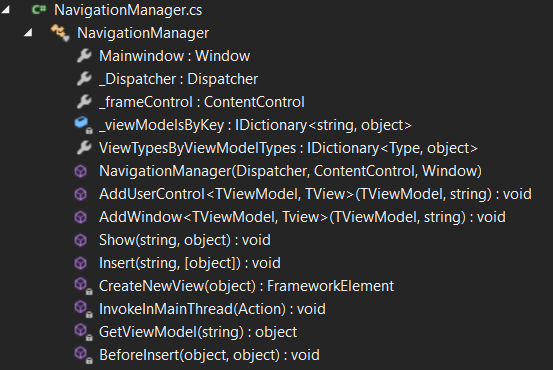


Рисунок 4.2 – Структура класса NavigationManager.cs

Свойство MainWindow предназначено для хранения ссылки на главное окно приложения.

Свойство \_Dispatcher предназначено для хранения текущего потока приложения.

Свойство \_frameControl предназначено для хранения контекста, в который будут вставляться окна приложенияя.

Поле \_viewModelsByKey представляет собой словарь, предназначенный для хранения данных в формате ключ-значение. В качестве ключа выступает строка, связанная с Моделью Представления, а в качестве значения – объект Модели Представления.

Свойство ViewTypesByViewModelTypes представляет собой словарь, ключом которого является тим Модели представления, а значение – объект Представления.

Метод AddUserControl предназначен для добавления данных о новом UserControl в словари \_viewModelsByKey и ViewTypesByViewModelTypes.

Метод AddWindow предназначен для добавления данных о новом окне в словари \_viewModelsByKey и ViewTypesByViewModelTypes.

Метод Show предназначен для отображения страницы в новом окне.

Метод Insert предназначен для вставки новой страницы в контекст окна.

Метод BeforeInsert предназначен для выполнения каких-либо действий перед вставкой новой страницы в окно.

Папка Resources предназначена для хранения ресурсов приложения. В данной папке находятся картинки-схемы, которые используются в приложении. Содержимое папки Resources представлено на рисунке 4.3.

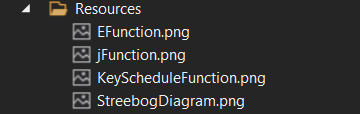


Рисунок 4.3 – Содержимое папки Resources

Папка StreebogAlgoritm содержит в себе два класса: InterimResults.cs и Streebog.cs.

Класс Streebog.cs содержит в себе методы и поля, которые необходимы для реализации алгоритма шифрования STREEBOG. Программный код класса Streebog.cs представлен в приложении А. Структура класса Streebog.cs представлена на рисунке 4.4.

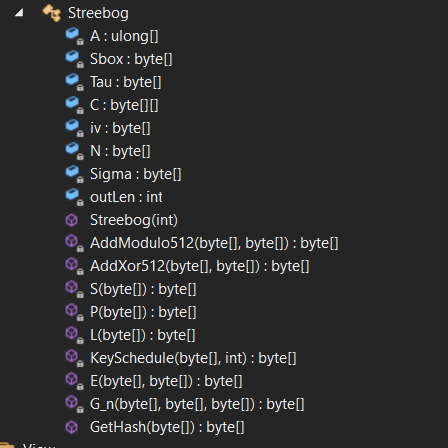


Рисунок 4.4 – Структура класса Streebog.cs

Поле А представляет собой массив шестнадцатеричных значений, который используется в операции линейного преобразования множество двоичных векторов.

Поле Sbox представляет собой массив значений подстановок, который используется в операции нелинейного биективного преобразования множества двоичных векторов.

Поле Tau представляет собой массив значений перестановок, который используется в операции перестановки байт.I

Поле iv представляет собой инициализирующий вектор, используемый в алгоритме.

Поле N и Sigma представляют собой вектора, используемы в работе алгоритма.

Поле outLen предназначено для хранения размера хеша.

Метод Streebog представляет собой конструктор, который необходим для создания объектов класса Streebog.cs. Входной параметр – целочисленное значение размера хеша.

Метод AddModulo512 предназначен для выполнения операция сложения по модулю 2512.

Метод АddXor512 предназначен для выполнения операции XOR над 512 битными массивами данных.

Метод S предназначен для выполнения операции нелинейного биективного преобразования множетсва двоичных векторов.

Метод P предназначен для выполнения операции перестановки байт.

Метод L предназначен для выполнения операции линейного преобразования множества двоичных векторов.

Метод KeySchedule предназначен для формирования раундовых ключей при работе блочного алгоритма шифрования.

Метод E представляет собой блочный алгоритм шифрования, используемы в алгоритме хеширования STREEBOG.

Метод G\_n представляет собой функцию сжатия, используемую в алгоритме хеширования STREEBOGЮ

Метод GetHash предназначен для получения хеша сообщения.

Класс InterimResults.cs предназначен для хранения промежуточных значений при работе алгоритма хеширования.

Папка View предназначен для хранения представлений окон, которые используется в работе приложения.

Папка ViewModel предназначена для хранения моделей представлений, которые используется в работе приложения.

# **Тестирование программного средства**

Выполним тестирование разработанного программного средства. Для этого осуществим запуск программы и убедимся в успешном открытии стартового окна. Вид стартового окна представлен на рисунке 6.1.

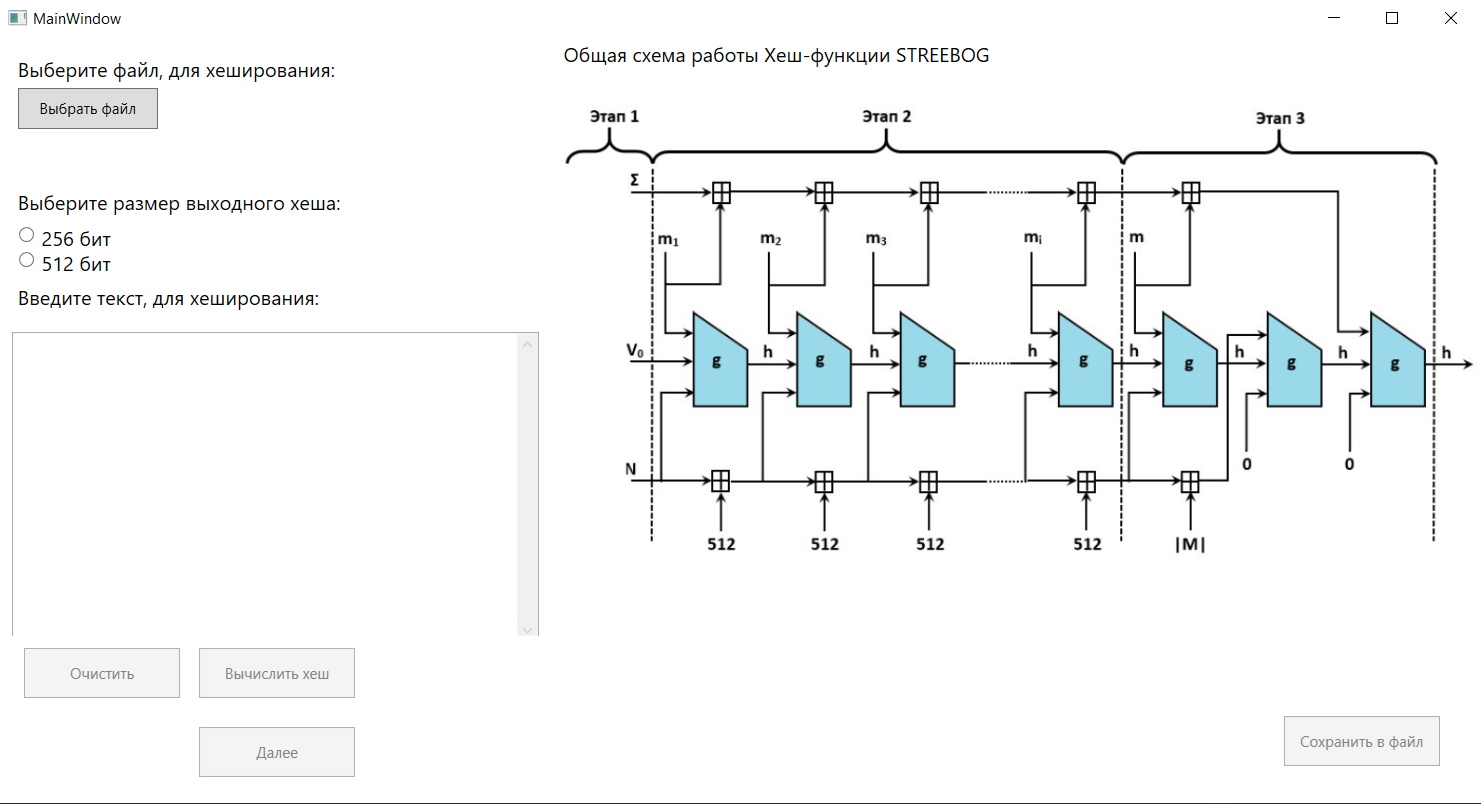


Рисунок 6.1 – Стартовое окно программы

Как видно из рисунка 6.1 программа запустилось успешно. Проверим на контрольном примере хеширование сообщения. Для этого необходимо выбрать файл для хеширования, нажав на кнопку “Выбрать файл”, после чего отобразиться окно для выбора файла, представленное на рисунке 6.2.

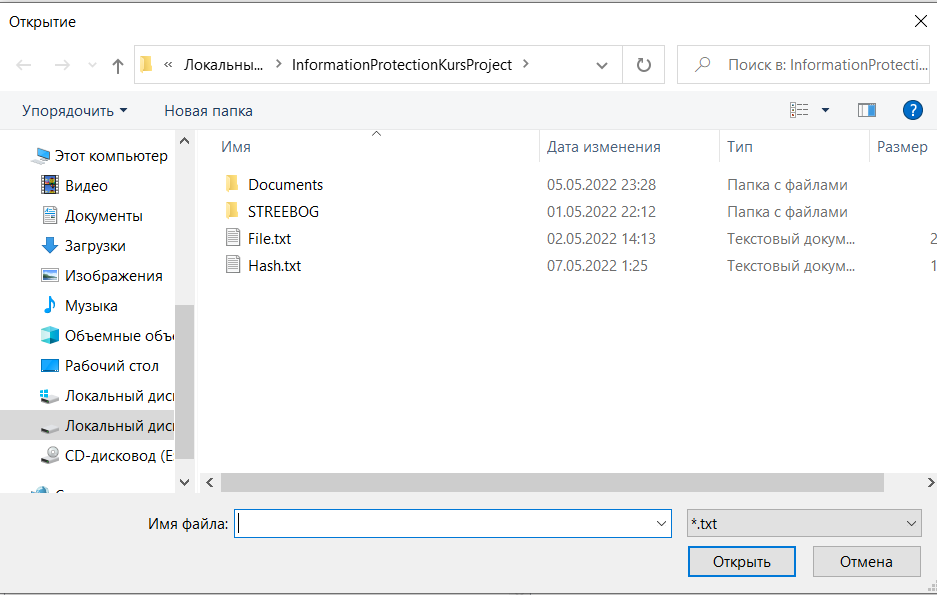


Рисунок 6.2 – Окно выбора файла

Также можно было ввести текст напрямую в текстовое поле. После выбора файла, необходимо выбрать длину хеша. Для этого предоставлены две радио-кнопки со значениями в 256 и 512 бит соответственно. После выбора параметров стартовое окно будет иметь вид, представленный на рисунке 6.3.

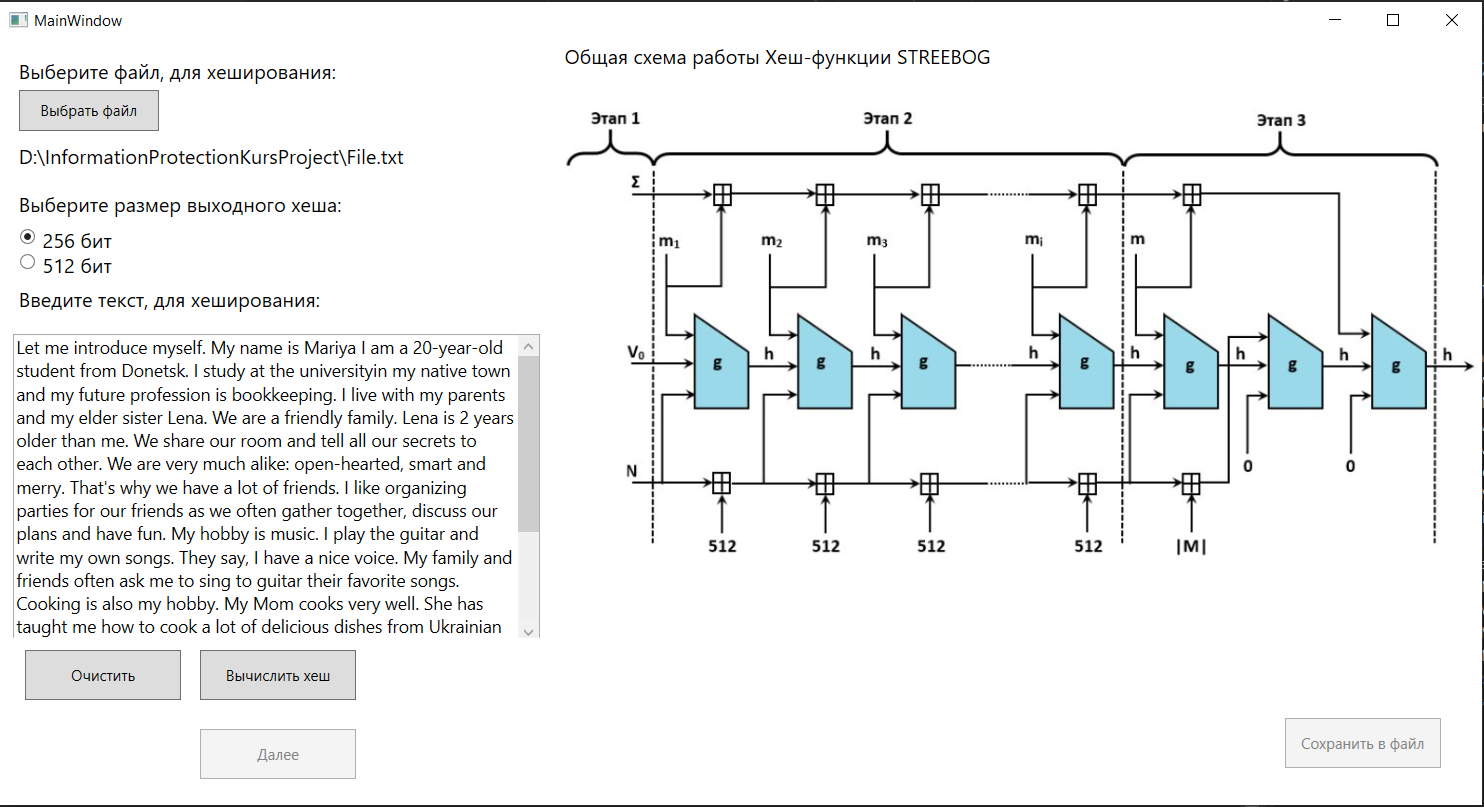


Рисунок 6.3 – Стартовое окно программы, после выбора параметров хеширования

Далее выполним нажатие на кнопку “Вычислить хеш”, после чего программа должна успешно вычислить хеш сообщения и отобразить соответствующее окно с результатом хеширования (рисунок 6.4).

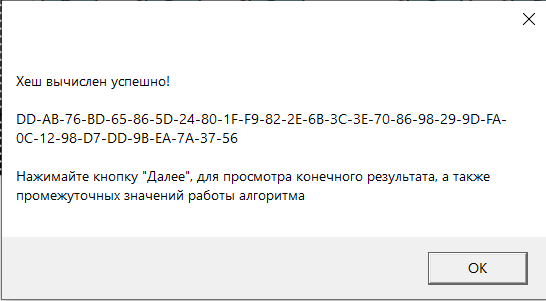


Рисунок 6.4 – Окно с результатом хеширования

На следующем шаге протестируем кнопку записи результата хеширования в текстовый файл. Для этого осуществим нажатие на кнопку “Сохранить в файл”. В окне выбора файла (рисунок 6.2) изберем файл, в который сохраним результат хеширования. После выполнения операций получим сообщение о результате сохранения (рисунок 6.5).

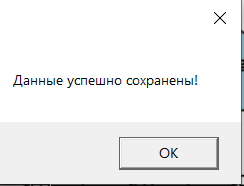


Рисунок 6.5 – Окно о успешном сохранении данных

Также получим текстовый файл с результатом хеширования представленый на рисунке 6.6.

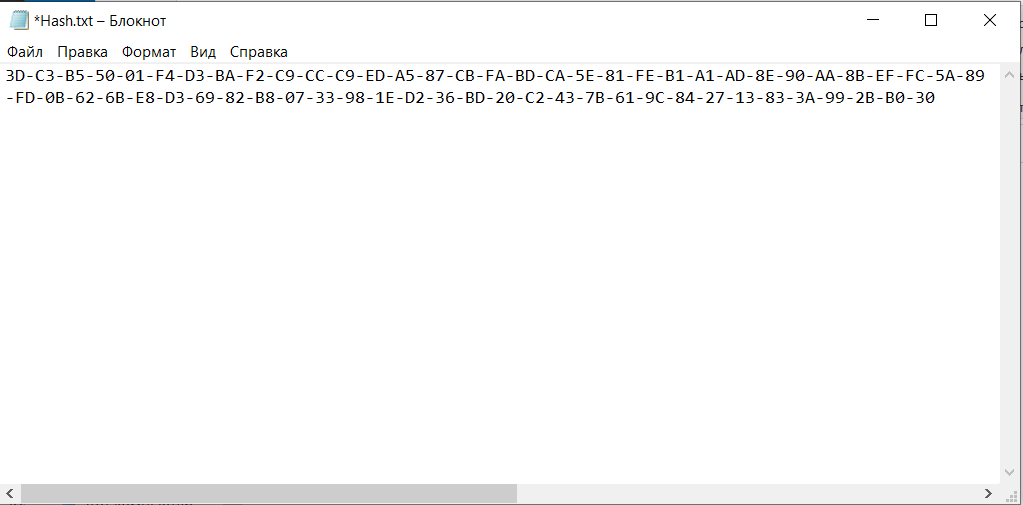


Рисунок 6.6 – Файл с результатом хеширования

Далее протестируем программу на сброс параметров хеширования. Для этого осуществим нажатие на кнопку “Очистить”. Результат выполнения данной операции представлен на рисунке 6.7.

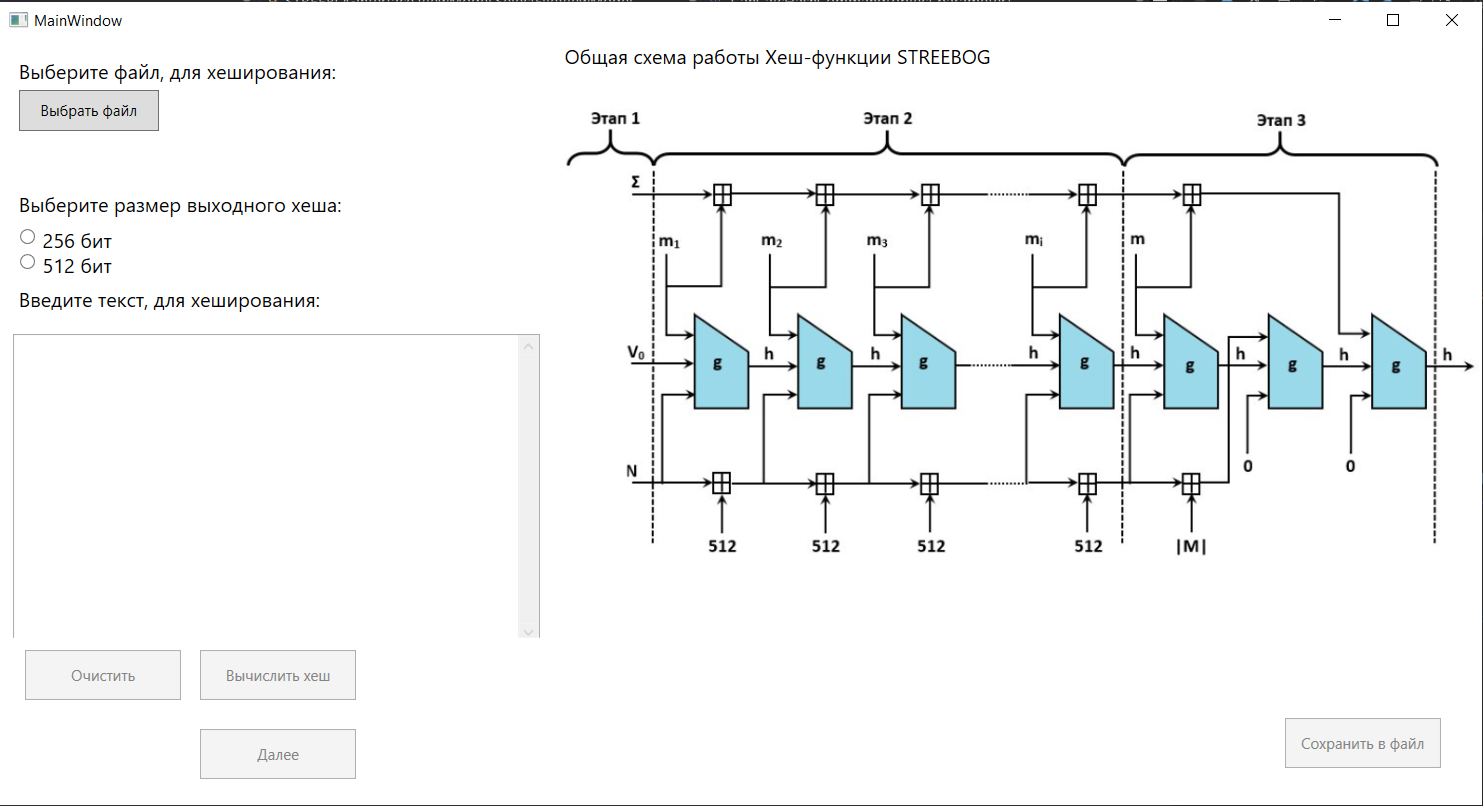


Рисунок 6.7 – Сброс параметров программы

Из рисунка 6.7 видно, что сброс параметров программы произошел успешно. Этому свидетельствует отсутствие имени раннее выбранного файла, а также несигнальное состояние радио - кнопок и отсутствие текста в текстовом блоке.

Перейдем к тестированию смены страниц в приложении. Для получения следующей страницы предназначена кнопка “Далее”. Осуществим нажатие на данную кнопку и зафиксируем полученный результат (рисунок 6.8).

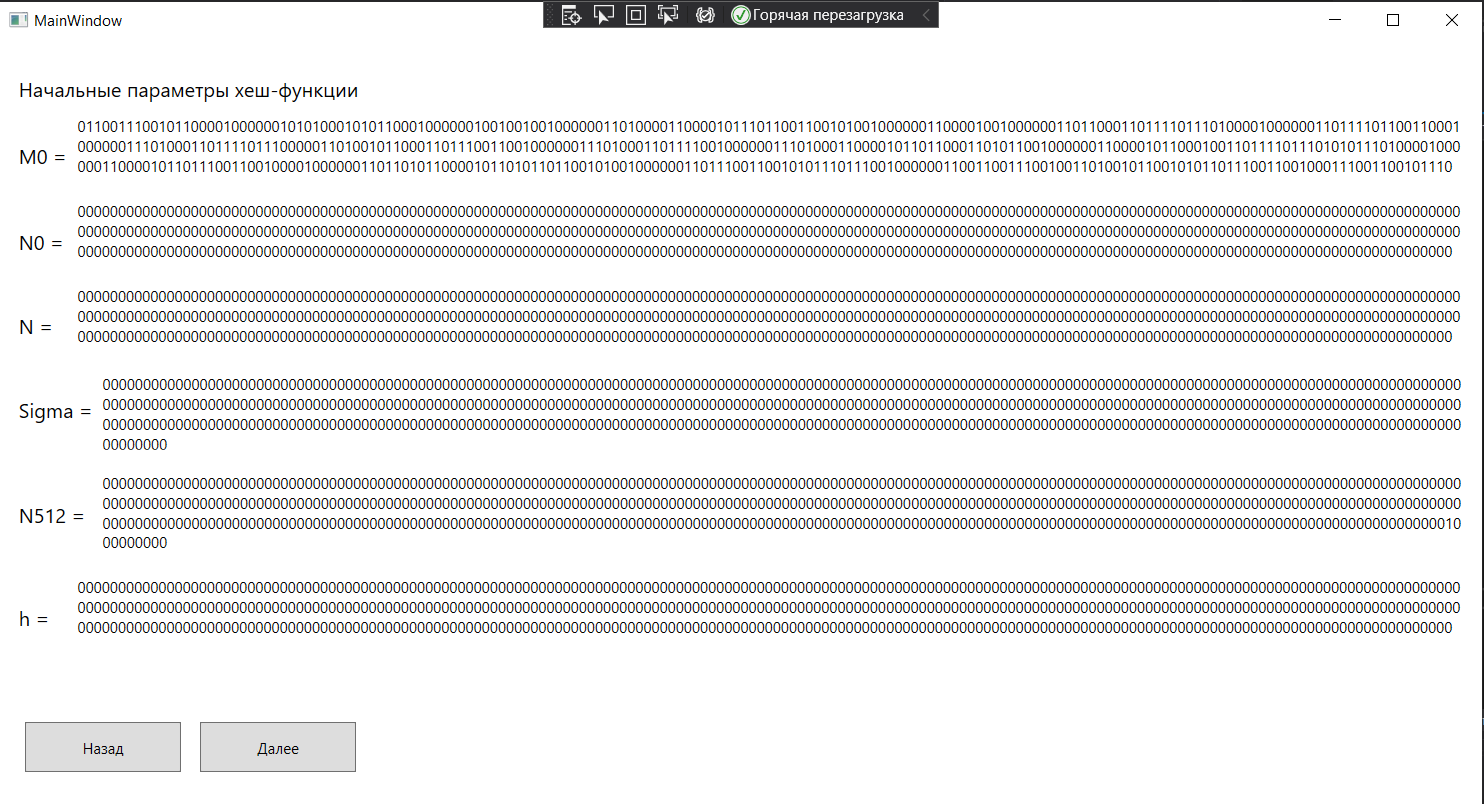


Рисунок 6.8 – Результат нажатия на кнопку “Далее”

Из рисунка 6.8 можно увидеть, что операция смены страницы выполнилась успешно, т.к. отобразилась следующая страница с результатами работы алгоритма хеширования.

Для получения предыдущей страницы предназначена кнопка “Назад”. Осуществим нажатие на данную кнопку и зафиксируем полученный результат (рисунок 6.9).

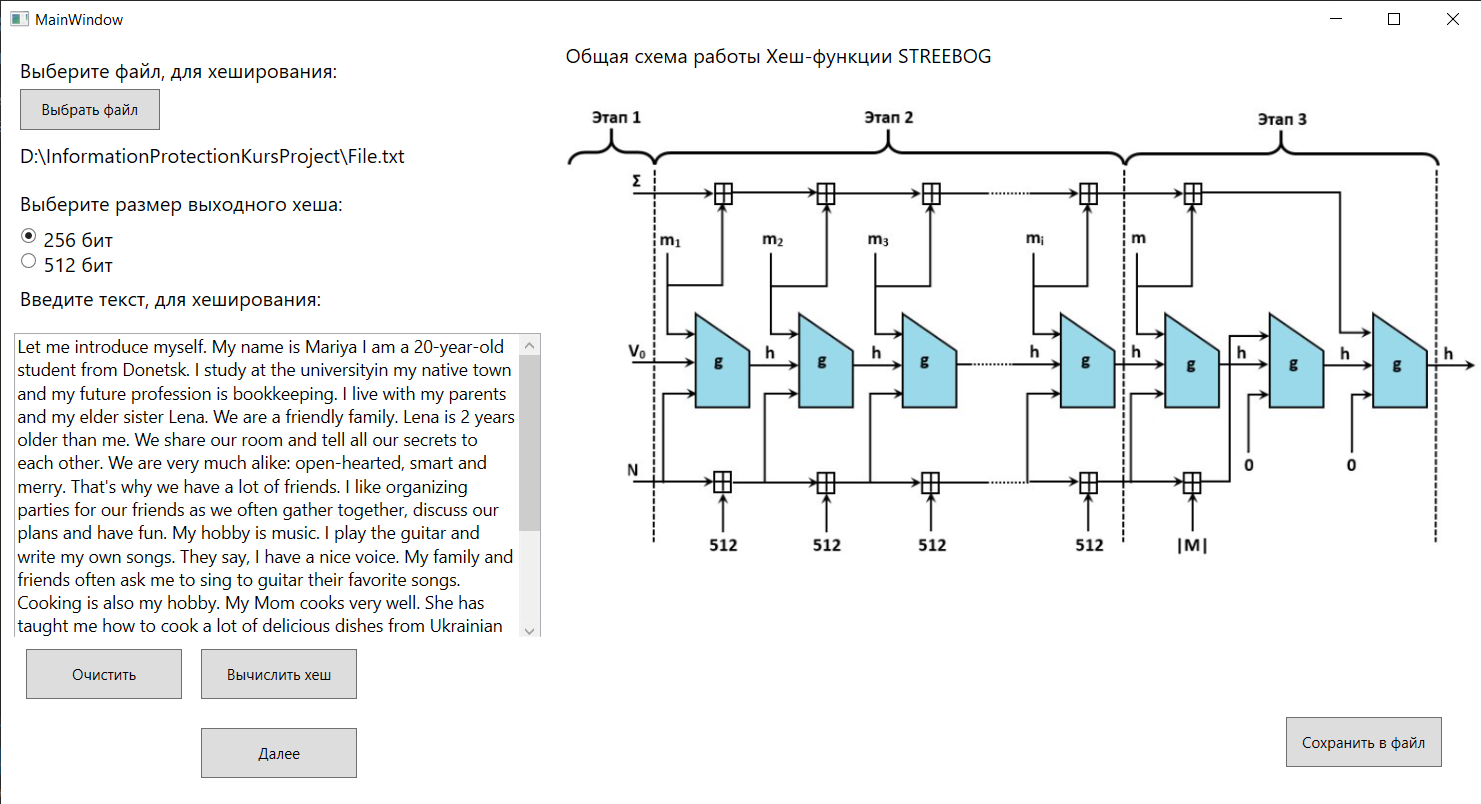


Рисунок 6.9 – Результат нажатия на кнопку “Далее”

Из рисунка 6.9 можно увидеть, что операция смены страницы выполнилась успешно, т.к. отобразилась предыдущая страница программы.

В результате проведения тестирования можно сказать, что разработанная программа успешно функционирует.

# **Руководство пользователя**

После выполнения запуска программы будет открыта стартовая страница, которая представлена на рисунке 7.1

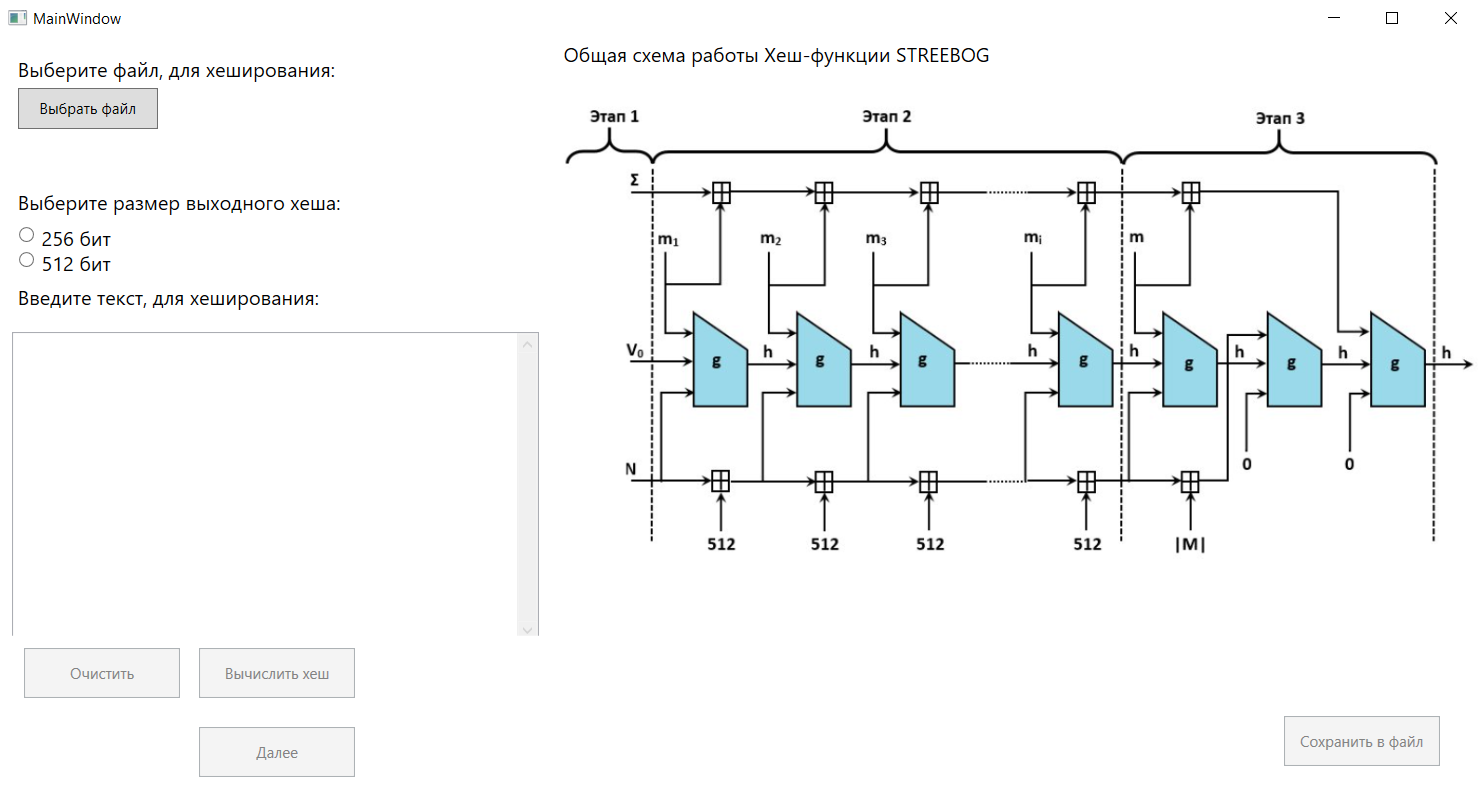


Рисунок 7.1 – Стартовая страница программы

Ознакомимся с элементами, размещенными на стартовой странице. Кнопка “Выбрать файл” предназначена для выбора файла для хеширования. Радио-кнопки предназначены для выбора длины хеша сообщения. Текстовый блок предназначен для ввода сообщения вручную, также в нем отображается текст файла, если он заранее был выбран. Кнопка “Очистить” предназначена для сброса всех выбранных параметров. Кнопка “Вычислить хеш” предназначена для вычисления хеша сообщения. Кнопка “Далее” предназначена для отображения следующей страницы приложения. Кнопка “Сохранить в файл” предназначена для сохранения результатов работы хеш-функции в текстовый файл.

Для того, что бы выполнить хеширование сообщения необходимо загрузить файл либо вручную ввести текст в текстовое поле, также необходимо выбрать длину получаемого хеша 256 или 512 бит. Если данные параметры не будут установлены, то кнопка “Вычислить хеш” не перейдет в активное состояние. Если же все условия выполнены стартовая странице примет вид, представленный на рисунке 7.2

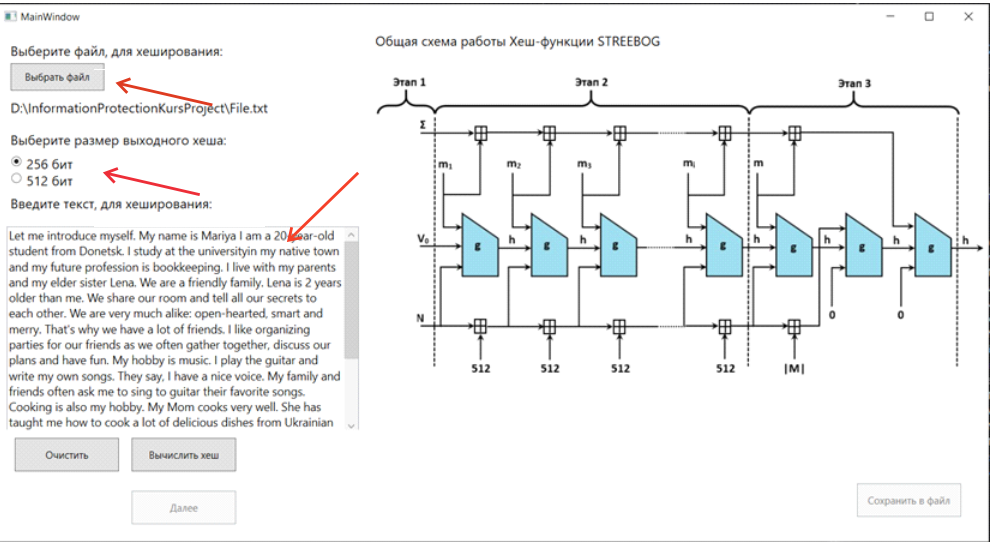


Рисунок 7.2 – Стартовое окно программы, после выбора параметров хеширования

После установки параметров следует нажать на кнопку “Вычислить хеш”, для того, чтобы получить хеш исходного сообщения. Кнопка показана на рисунке 7.2.

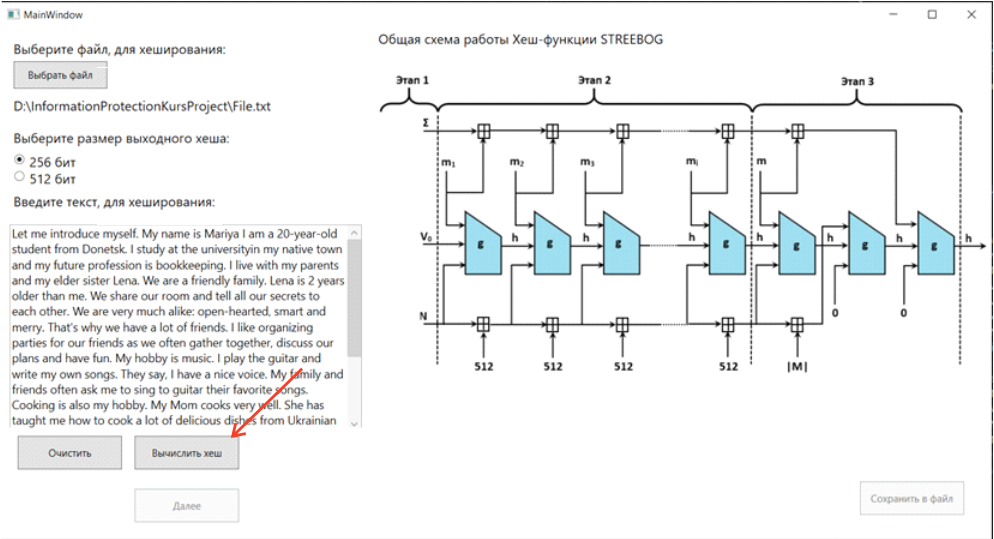


Рисунок 7.3 – Нажатие на кнопку «Вычислить хеш»

После того, как хеш будет вычислен, появится соответствующее сообщения с результатом и кнопки “Далее” и “Сохранить в файл” станут доступными для использования. Следует сохранить полученный результат, во избежание его потери, нажав кнопку “Сохранить в файл” и выбрав файл, для сохранения. Кнопка показана на рисунке 7.4

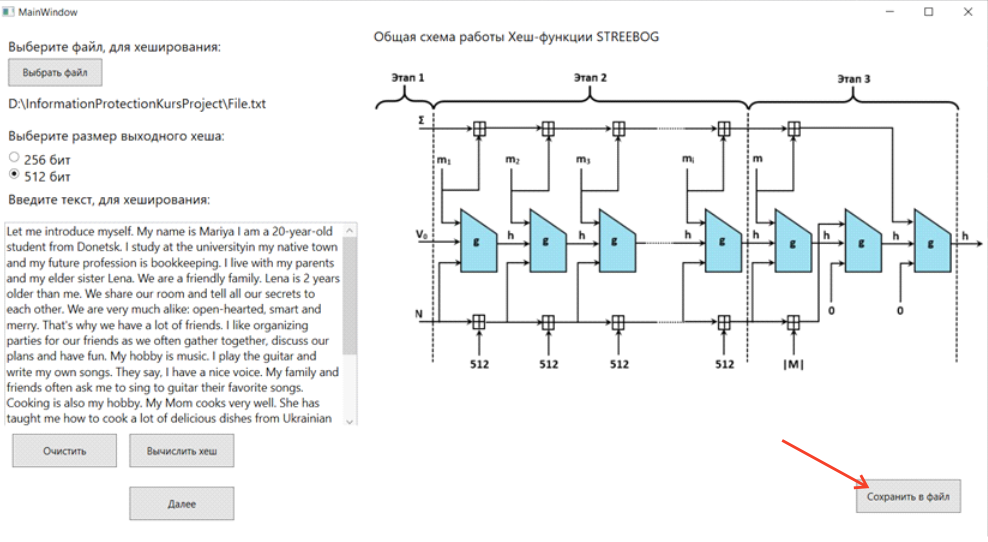


Рисунок 7.4 – Нажатие на кнопку «Сохранить в файл»

После того, как результат буде сохранен, появится соответствующее сообщение и файл с результатом хеширования появится на диске.

Затем стоит просмотреть промежуточные результаты работы алгоритма. Для этого предназначены две кнопки “Назад” и “Далее ” по нажатию на которые будет открываться соответственно предыдущее или следующее окно. Кнопки показаны на рисунке 7.5



Рисунок 7.5 – Нажатие на кнопку «Назад» или «Далее»

Для выхода из программы необходимо нажать на крестик в правом верхнем углу окна приложения, как показано на рисунке 7.6



Рисунок 7.6 – Выход из приложения

Таким образом в данном разделе было составлено руководство пользователя, которое нацелено на упрощение работы с разработанным программным обеспечением.

# **Заключение**

В результате выполнения данной курсовой работы было разработано программное обеспечение, позволяющее вычислять хеш сообщения по алгоритму STREEBOG.

Программа предоставляет следующие возможности:

* вычисление хеша входных данных при помощи алгоритма хеширования STREEBOG;
* возможность ввода данных, необходимых для вычисления хеша, путем заполнения текстового блока;
* возможность ввода данных, необходимых для вычисления хеша, путем взятия их из текстового файла;
* выбор конечного размера хеш-кода(512 или 256 бит).
* отображение результата вычисления хеша сообщения;
* запись результата вычисления хеша в текстовый файл.
* отображение результаты работы промежуточных частей работы алгоритма хеширования.

Алгоритм хеширования STREEBOG применяется в основном, для формирования электронно-цифровых подписей. Алгоритм обладает рядом свойств, которые соответствуют современным требованиям к криптографической стойкости и которые не позволяют успешно применять известные на сегодняшний день атаки. Также данный алгоритм весьма эффективен и скорость формирование хеш-кода достаточно высока, что делает его одним из лучших алгоритмов хеширования на сегодняшний день.

# **Список используемых источников**

1. Википедия [Электронный ресурс] / Wikimedia Foundation Inc. – 2020. – Режим доступа: https://ru.wikipedia.org/wiki/Структура\_Меркла\_—\_Дамгора. – Дата доступа: 07.05.2022
2. Википедия [Электронный ресурс] / Wikimedia Foundation Inc. – 2020. – Режим доступа: https://ru.wikipedia.org/wiki/Функция\_Миагути\_—\_Пренеля. – Дата доступа: 07.05.2022
3. ГOСТ P 34.11-2012. Криптографическая защита информации. – Взамен ГОСТ Р 34.11-94 – Введ 2013.01.01 – 15c
4. Понамарев Вячеслав Программирование на C++/C# в Visual Studio .NET 2003; БХВ-Петербург – М., 2004. – 352 c.
5. Культин Никита Основы программирования в Microsoft Visual C# 2010; БХВ-Петербург – М., 2011. – 634 c.

# **ПРИЛОЖЕНИЕ А**

Streebog.cs

|  |
| --- |
| public class Streebog  {  private ulong[] A = {  0x8e20faa72ba0b470, 0x47107ddd9b505a38, 0xad08b0e0c3282d1c, 0xd8045870ef14980e,  0x6c022c38f90a4c07, 0x3601161cf205268d, 0x1b8e0b0e798c13c8, 0x83478b07b2468764,  0xa011d380818e8f40, 0x5086e740ce47c920, 0x2843fd2067adea10, 0x14aff010bdd87508,  0x0ad97808d06cb404, 0x05e23c0468365a02, 0x8c711e02341b2d01, 0x46b60f011a83988e,  0x90dab52a387ae76f, 0x486dd4151c3dfdb9, 0x24b86a840e90f0d2, 0x125c354207487869,  0x092e94218d243cba, 0x8a174a9ec8121e5d, 0x4585254f64090fa0, 0xaccc9ca9328a8950,  0x9d4df05d5f661451, 0xc0a878a0a1330aa6, 0x60543c50de970553, 0x302a1e286fc58ca7,  0x18150f14b9ec46dd, 0x0c84890ad27623e0, 0x0642ca05693b9f70, 0x0321658cba93c138,  0x86275df09ce8aaa8, 0x439da0784e745554, 0xafc0503c273aa42a, 0xd960281e9d1d5215,  0xe230140fc0802984, 0x71180a8960409a42, 0xb60c05ca30204d21, 0x5b068c651810a89e,  0x456c34887a3805b9, 0xac361a443d1c8cd2, 0x561b0d22900e4669, 0x2b838811480723ba,  0x9bcf4486248d9f5d, 0xc3e9224312c8c1a0, 0xeffa11af0964ee50, 0xf97d86d98a327728,  0xe4fa2054a80b329c, 0x727d102a548b194e, 0x39b008152acb8227, 0x9258048415eb419d,  0x492c024284fbaec0, 0xaa16012142f35760, 0x550b8e9e21f7a530, 0xa48b474f9ef5dc18,  0x70a6a56e2440598e, 0x3853dc371220a247, 0x1ca76e95091051ad, 0x0edd37c48a08a6d8,  0x07e095624504536c, 0x8d70c431ac02a736, 0xc83862965601dd1b, 0x641c314b2b8ee083  };  private byte[] Sbox ={  0xFC, 0xEE, 0xDD, 0x11, 0xCF, 0x6E, 0x31, 0x16, 0xFB, 0xC4, 0xFA, 0xDA, 0x23, 0xC5, 0x04, 0x4D,  0xE9, 0x77, 0xF0, 0xDB, 0x93, 0x2E, 0x99, 0xBA, 0x17, 0x36, 0xF1, 0xBB, 0x14, 0xCD, 0x5F, 0xC1,  0xF9, 0x18, 0x65, 0x5A, 0xE2, 0x5C, 0xEF, 0x21, 0x81, 0x1C, 0x3C, 0x42, 0x8B, 0x01, 0x8E, 0x4F,  0x05, 0x84, 0x02, 0xAE, 0xE3, 0x6A, 0x8F, 0xA0, 0x06, 0x0B, 0xED, 0x98, 0x7F, 0xD4, 0xD3, 0x1F,  0xEB, 0x34, 0x2C, 0x51, 0xEA, 0xC8, 0x48, 0xAB, 0xF2, 0x2A, 0x68, 0xA2, 0xFD, 0x3A, 0xCE, 0xCC,  0xB5, 0x70, 0x0E, 0x56, 0x08, 0x0C, 0x76, 0x12, 0xBF, 0x72, 0x13, 0x47, 0x9C, 0xB7, 0x5D, 0x87,  0x15, 0xA1, 0x96, 0x29, 0x10, 0x7B, 0x9A, 0xC7, 0xF3, 0x91, 0x78, 0x6F, 0x9D, 0x9E, 0xB2, 0xB1,  0x32, 0x75, 0x19, 0x3D, 0xFF, 0x35, 0x8A, 0x7E, 0x6D, 0x54, 0xC6, 0x80, 0xC3, 0xBD, 0x0D, 0x57,  0xDF, 0xF5, 0x24, 0xA9, 0x3E, 0xA8, 0x43, 0xC9, 0xD7, 0x79, 0xD6, 0xF6, 0x7C, 0x22, 0xB9, 0x03,  0xE0, 0x0F, 0xEC, 0xDE, 0x7A, 0x94, 0xB0, 0xBC, 0xDC, 0xE8, 0x28, 0x50, 0x4E, 0x33, 0x0A, 0x4A,  0xA7, 0x97, 0x60, 0x73, 0x1E, 0x00, 0x62, 0x44, 0x1A, 0xB8, 0x38, 0x82, 0x64, 0x9F, 0x26, 0x41,  0xAD, 0x45, 0x46, 0x92, 0x27, 0x5E, 0x55, 0x2F, 0x8C, 0xA3, 0xA5, 0x7D, 0x69, 0xD5, 0x95, 0x3B,  0x07, 0x58, 0xB3, 0x40, 0x86, 0xAC, 0x1D, 0xF7, 0x30, 0x37, 0x6B, 0xE4, 0x88, 0xD9, 0xE7, 0x89,  0xE1, 0x1B, 0x83, 0x49, 0x4C, 0x3F, 0xF8, 0xFE, 0x8D, 0x53, 0xAA, 0x90, 0xCA, 0xD8, 0x85, 0x61,  0x20, 0x71, 0x67, 0xA4, 0x2D, 0x2B, 0x09, 0x5B, 0xCB, 0x9B, 0x25, 0xD0, 0xBE, 0xE5, 0x6C, 0x52,  0x59, 0xA6, 0x74, 0xD2, 0xE6, 0xF4, 0xB4, 0xC0, 0xD1, 0x66, 0xAF, 0xC2, 0x39, 0x4B, 0x63, 0xB6  };  private byte[] Tau ={  0, 8, 16, 24, 32, 40, 48, 56,  1, 9, 17, 25, 33, 41, 49, 57,  2, 10, 18, 26, 34, 42, 50, 58,  3, 11, 19, 27, 35, 43, 51, 59,  4, 12, 20, 28, 36, 44, 52, 60,  5, 13, 21, 29, 37, 45, 53, 61,  6, 14, 22, 30, 38, 46, 54, 62,  7, 15, 23, 31, 39, 47, 55, 63  };  private byte[][] C = {  new byte[64]{  0xb1,0x08,0x5b,0xda,0x1e,0xca,0xda,0xe9,0xeb,0xcb,0x2f,0x81,0xc0,0x65,0x7c,0x1f,  0x2f,0x6a,0x76,0x43,0x2e,0x45,0xd0,0x16,0x71,0x4e,0xb8,0x8d,0x75,0x85,0xc4,0xfc,  0x4b,0x7c,0xe0,0x91,0x92,0x67,0x69,0x01,0xa2,0x42,0x2a,0x08,0xa4,0x60,0xd3,0x15,  0x05,0x76,0x74,0x36,0xcc,0x74,0x4d,0x23,0xdd,0x80,0x65,0x59,0xf2,0xa6,0x45,0x07  },  new byte[64]{  0x6f,0xa3,0xb5,0x8a,0xa9,0x9d,0x2f,0x1a,0x4f,0xe3,0x9d,0x46,0x0f,0x70,0xb5,0xd7,  0xf3,0xfe,0xea,0x72,0x0a,0x23,0x2b,0x98,0x61,0xd5,0x5e,0x0f,0x16,0xb5,0x01,0x31,  0x9a,0xb5,0x17,0x6b,0x12,0xd6,0x99,0x58,0x5c,0xb5,0x61,0xc2,0xdb,0x0a,0xa7,0xca,  0x55,0xdd,0xa2,0x1b,0xd7,0xcb,0xcd,0x56,0xe6,0x79,0x04,0x70,0x21,0xb1,0x9b,0xb7  },  new byte[64]{  0xf5,0x74,0xdc,0xac,0x2b,0xce,0x2f,0xc7,0x0a,0x39,0xfc,0x28,0x6a,0x3d,0x84,0x35,  0x06,0xf1,0x5e,0x5f,0x52,0x9c,0x1f,0x8b,0xf2,0xea,0x75,0x14,0xb1,0x29,0x7b,0x7b,  0xd3,0xe2,0x0f,0xe4,0x90,0x35,0x9e,0xb1,0xc1,0xc9,0x3a,0x37,0x60,0x62,0xdb,0x09,  0xc2,0xb6,0xf4,0x43,0x86,0x7a,0xdb,0x31,0x99,0x1e,0x96,0xf5,0x0a,0xba,0x0a,0xb2  },  new byte[64]{  0xef,0x1f,0xdf,0xb3,0xe8,0x15,0x66,0xd2,0xf9,0x48,0xe1,0xa0,0x5d,0x71,0xe4,0xdd,  0x48,0x8e,0x85,0x7e,0x33,0x5c,0x3c,0x7d,0x9d,0x72,0x1c,0xad,0x68,0x5e,0x35,0x3f,  0xa9,0xd7,0x2c,0x82,0xed,0x03,0xd6,0x75,0xd8,0xb7,0x13,0x33,0x93,0x52,0x03,0xbe,  0x34,0x53,0xea,0xa1,0x93,0xe8,0x37,0xf1,0x22,0x0c,0xbe,0xbc,0x84,0xe3,0xd1,0x2e  },  new byte[64]{  0x4b,0xea,0x6b,0xac,0xad,0x47,0x47,0x99,0x9a,0x3f,0x41,0x0c,0x6c,0xa9,0x23,0x63,  0x7f,0x15,0x1c,0x1f,0x16,0x86,0x10,0x4a,0x35,0x9e,0x35,0xd7,0x80,0x0f,0xff,0xbd,  0xbf,0xcd,0x17,0x47,0x25,0x3a,0xf5,0xa3,0xdf,0xff,0x00,0xb7,0x23,0x27,0x1a,0x16,  0x7a,0x56,0xa2,0x7e,0xa9,0xea,0x63,0xf5,0x60,0x17,0x58,0xfd,0x7c,0x6c,0xfe,0x57  },  new byte[64]{  0xae,0x4f,0xae,0xae,0x1d,0x3a,0xd3,0xd9,0x6f,0xa4,0xc3,0x3b,0x7a,0x30,0x39,0xc0,  0x2d,0x66,0xc4,0xf9,0x51,0x42,0xa4,0x6c,0x18,0x7f,0x9a,0xb4,0x9a,0xf0,0x8e,0xc6,  0xcf,0xfa,0xa6,0xb7,0x1c,0x9a,0xb7,0xb4,0x0a,0xf2,0x1f,0x66,0xc2,0xbe,0xc6,0xb6,  0xbf,0x71,0xc5,0x72,0x36,0x90,0x4f,0x35,0xfa,0x68,0x40,0x7a,0x46,0x64,0x7d,0x6e  },  new byte[64]{  0xf4,0xc7,0x0e,0x16,0xee,0xaa,0xc5,0xec,0x51,0xac,0x86,0xfe,0xbf,0x24,0x09,0x54,  0x39,0x9e,0xc6,0xc7,0xe6,0xbf,0x87,0xc9,0xd3,0x47,0x3e,0x33,0x19,0x7a,0x93,0xc9,  0x09,0x92,0xab,0xc5,0x2d,0x82,0x2c,0x37,0x06,0x47,0x69,0x83,0x28,0x4a,0x05,0x04,  0x35,0x17,0x45,0x4c,0xa2,0x3c,0x4a,0xf3,0x88,0x86,0x56,0x4d,0x3a,0x14,0xd4,0x93  },  new byte[64]{  0x9b,0x1f,0x5b,0x42,0x4d,0x93,0xc9,0xa7,0x03,0xe7,0xaa,0x02,0x0c,0x6e,0x41,0x41,  0x4e,0xb7,0xf8,0x71,0x9c,0x36,0xde,0x1e,0x89,0xb4,0x44,0x3b,0x4d,0xdb,0xc4,0x9a,  0xf4,0x89,0x2b,0xcb,0x92,0x9b,0x06,0x90,0x69,0xd1,0x8d,0x2b,0xd1,0xa5,0xc4,0x2f,  0x36,0xac,0xc2,0x35,0x59,0x51,0xa8,0xd9,0xa4,0x7f,0x0d,0xd4,0xbf,0x02,0xe7,0x1e  },  new byte[64]{  0x37,0x8f,0x5a,0x54,0x16,0x31,0x22,0x9b,0x94,0x4c,0x9a,0xd8,0xec,0x16,0x5f,0xde,  0x3a,0x7d,0x3a,0x1b,0x25,0x89,0x42,0x24,0x3c,0xd9,0x55,0xb7,0xe0,0x0d,0x09,0x84,  0x80,0x0a,0x44,0x0b,0xdb,0xb2,0xce,0xb1,0x7b,0x2b,0x8a,0x9a,0xa6,0x07,0x9c,0x54,  0x0e,0x38,0xdc,0x92,0xcb,0x1f,0x2a,0x60,0x72,0x61,0x44,0x51,0x83,0x23,0x5a,0xdb  },  new byte[64]{  0xab,0xbe,0xde,0xa6,0x80,0x05,0x6f,0x52,0x38,0x2a,0xe5,0x48,0xb2,0xe4,0xf3,0xf3,  0x89,0x41,0xe7,0x1c,0xff,0x8a,0x78,0xdb,0x1f,0xff,0xe1,0x8a,0x1b,0x33,0x61,0x03,  0x9f,0xe7,0x67,0x02,0xaf,0x69,0x33,0x4b,0x7a,0x1e,0x6c,0x30,0x3b,0x76,0x52,0xf4,  0x36,0x98,0xfa,0xd1,0x15,0x3b,0xb6,0xc3,0x74,0xb4,0xc7,0xfb,0x98,0x45,0x9c,0xed  },  new byte[64]{  0x7b,0xcd,0x9e,0xd0,0xef,0xc8,0x89,0xfb,0x30,0x02,0xc6,0xcd,0x63,0x5a,0xfe,0x94,  0xd8,0xfa,0x6b,0xbb,0xeb,0xab,0x07,0x61,0x20,0x01,0x80,0x21,0x14,0x84,0x66,0x79,  0x8a,0x1d,0x71,0xef,0xea,0x48,0xb9,0xca,0xef,0xba,0xcd,0x1d,0x7d,0x47,0x6e,0x98,  0xde,0xa2,0x59,0x4a,0xc0,0x6f,0xd8,0x5d,0x6b,0xca,0xa4,0xcd,0x81,0xf3,0x2d,0x1b  },  new byte[64]{  0x37,0x8e,0xe7,0x67,0xf1,0x16,0x31,0xba,0xd2,0x13,0x80,0xb0,0x04,0x49,0xb1,0x7a,  0xcd,0xa4,0x3c,0x32,0xbc,0xdf,0x1d,0x77,0xf8,0x20,0x12,0xd4,0x30,0x21,0x9f,0x9b,  0x5d,0x80,0xef,0x9d,0x18,0x91,0xcc,0x86,0xe7,0x1d,0xa4,0xaa,0x88,0xe1,0x28,0x52,  0xfa,0xf4,0x17,0xd5,0xd9,0xb2,0x1b,0x99,0x48,0xbc,0x92,0x4a,0xf1,0x1b,0xd7,0x20  }  };  private byte[] iv = new byte[64];  private byte[] N = new byte[64];  private byte[] Sigma = new byte[64];  private int outLen = 0;  public Streebog(int outputLenght)  {  this.outLen = outputLenght;  if (this.outLen == 512)  {  for (int i = 0; i < 64; i++)  {  iv[i] = 0x00;  }  }  else if (this.outLen == 256)  {  for (int i = 0; i < 64; i++)  {  iv[i] = 0x01;  }  }  }  private byte[] AddModulo512(byte[] a, byte[] b)  {  byte[] temp = new byte[64];  int i = 0, t = 0;  byte[] tempA = new byte[64];  byte[] tempB = new byte[64];  Array.Copy(a, 0, tempA, 64 - a.Length, a.Length);  Array.Copy(b, 0, tempB, 64 - b.Length, b.Length);  InterimResults.N\_512.Add(InterimResults.ConvertToBinaryString(tempB));  for (i = 63; i >= 0; i--)  {  t = tempA[i] + tempB[i] + (t >> 8);  temp[i] = (byte)(t & 0xFF);  }  return temp;  }  private byte[] AddXor512(byte[] a, byte[] b)  {  byte[] c = new byte[64];  for (int i = 0; i < 64; i++)  c[i] = (byte)(a[i] ^ b[i]);  return c;  }  //Замена байтов  private byte[] S(byte[] state)  {  byte[] result = new byte[64];  for (int i = 0; i < 64; i++)  result[i] = Sbox[state[i]];  return result;  }  //Функция перестановки  private byte[] P(byte[] state)  {  byte[] result = new byte[64];  for (int i = 0; i < 64; i++)  {  result[i] = state[Tau[i]];  }  return result;  }  private byte[] L(byte[] state)  {  byte[] result = new byte[64];  for (int i = 0; i < 8; i++)  {  ulong t = 0;  byte[] tempArray = new byte[8];  //Делим 512 битную последовательность на 8 64-битных последовательностей  Array.Copy(state, i \* 8, tempArray, 0, 8);  InterimResults.GL64vector.Add(InterimResults.ConvertToBinaryString(tempArray));  tempArray = tempArray.Reverse().ToArray();  //Получаем массив битов из исходного массива байт  BitArray tempBits1 = new BitArray(tempArray);  bool[] tempBits = new bool[64];  //Переписываем его в логичесский массив, для последующего сравнения его значений  tempBits1.CopyTo(tempBits, 0);  tempBits = tempBits.Reverse().ToArray();  for (int j = 0; j < 64; j++)  {  if (tempBits[j] != false)  t = t ^ A[j];  }  byte[] ResPart = BitConverter.GetBytes(t).Reverse().ToArray();  InterimResults.LMatrixXorResult.Add(InterimResults.ConvertToBinaryString(ResPart));  Array.Copy(ResPart, 0, result, i \* 8, 8);  }  return result;  }  private byte[] KeySchedule(byte[] K, int i)  {  InterimResults.KeyScheduleKparameter.Add(InterimResults.ConvertToBinaryString(K));  InterimResults.KeyScheduleCparameter.Add(InterimResults.ConvertToBinaryString(C[i]));  //Слаживаем значение K с соответствующим значением из матрицы C  K = AddXor512(K, C[i]);  InterimResults.KxorC.Add(InterimResults.ConvertToBinaryString(K));  K = S(K);  InterimResults.KeyScheduleSresult.Add(InterimResults.ConvertToBinaryString(K));  K = P(K);  InterimResults.KeySchedulePresult.Add(InterimResults.ConvertToBinaryString(K));  K = L(K);  InterimResults.KeyScheduleLresult.Add(InterimResults.ConvertToBinaryString(K));  return K;  }  private byte[] E(byte[] K, byte[] m)  {  InterimResults.EKparameter.Add(InterimResults.ConvertToBinaryString(K));  byte[] state = AddXor512(K, m);  InterimResults.KxorM.Add(InterimResults.ConvertToBinaryString(state));  for (int i = 0; i < 12; i++)  {  state = S(state);  InterimResults.ESresult.Add(InterimResults.ConvertToBinaryString(state));  state = P(state);  InterimResults.EPresult.Add(InterimResults.ConvertToBinaryString(state));  state = L(state);  InterimResults.ELresult.Add(InterimResults.ConvertToBinaryString(state));  K = KeySchedule(K, i);  InterimResults.KeyScheduleResult.Add(InterimResults.ConvertToBinaryString(K));  state = AddXor512(state, K);  InterimResults.LxorKeySchedule.Add(InterimResults.ConvertToBinaryString(state));  }  return state;  }  //Функция сжатия  private byte[] G\_n(byte[] N, byte[] h, byte[] m)  {  InterimResults.N.Add(InterimResults.ConvertToBinaryString(N));  InterimResults.H.Add(InterimResults.ConvertToBinaryString(h));  InterimResults.initialMessage.Add(InterimResults.ConvertToBinaryString(m));  byte[] K = AddXor512(h, N);  InterimResults.NxorH.Add(InterimResults.ConvertToBinaryString(K));  K = S(K);  InterimResults.GSresult.Add(InterimResults.ConvertToBinaryString(K));  K = P(K);  InterimResults.GPresult.Add(InterimResults.ConvertToBinaryString(K));  K = L(K);  InterimResults.GLresult.Add(InterimResults.ConvertToBinaryString(K));  byte[] t = E(K, m);  InterimResults.EFunctionResult.Add(InterimResults.ConvertToBinaryString(t));  t = AddXor512(t, h);  byte[] newh = AddXor512(t, m);  InterimResults.EFunctionXORResult.Add(InterimResults.ConvertToBinaryString(newh));  return newh;  }  public byte[] GetHash(byte[] message)  {  byte[] paddedMes = new byte[64];  //Длина исходного сообщения в битах  int len = message.Length \* 8;  //Хэш  byte[] h = new byte[64];  byte[] N\_0 ={  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00  };  Array.Copy(iv, N, 64);  Array.Copy(iv, Sigma, 64);  byte[] N\_512 = BitConverter.GetBytes(512);  InterimResults.N\_0 = InterimResults.ConvertToBinaryString(N\_0);  InterimResults.Sigma.Add(InterimResults.ConvertToBinaryString(Sigma));  int inc = 0;  while (len >= 512)  {  inc++;  byte[] tempMes = new byte[64];  Array.Copy(message, message.Length - inc \* 64, tempMes, 0, 64);  //Вызываем функцию сжатия  h = G\_n(N, h, tempMes);  //Вызываем функцию, для вычисления следующего N  N = AddModulo512(N, N\_512.Reverse().ToArray());  //Вызываем функцию, для вычисления следующего Sigma (не разобрал функцию сложения по модулю)  Sigma = AddModulo512(Sigma, tempMes);  InterimResults.Sigma.Add(InterimResults.ConvertToBinaryString(Sigma));  len -= 512;  }  //Выделяем блок сообщения, который меньше 512 бит, для последующего расширения  byte[] message1 = new byte[message.Length - inc \* 64];  Array.Copy(message, 0, message1, 0, message.Length - inc \* 64);  InterimResults.SmallBlock = InterimResults.ConvertToBinaryString(message1);  //добавляем до нужного размера  if (message1.Length < 64)  {  for (int i = 0; i < (64 - message1.Length - 1); i++)  {  //Формируем массив из нулей  paddedMes[i] = 0;  }  paddedMes[64 - message1.Length - 1] = 0x01;  //Соединяем 2 массива(расширили блок до 512 бит)  Array.Copy(message1, 0, paddedMes, 64 - message1.Length, message1.Length);  InterimResults.ExtensionBlock = InterimResults.ConvertToBinaryString(paddedMes);  }  //Вызываем функцию сжатия  h = G\_n(N, h, paddedMes);  InterimResults.H.Add(InterimResults.ConvertToBinaryString(h));  //Что делается дальше?  byte[] MesLen = BitConverter.GetBytes(message1.Length \* 8);  //3 этап  N = AddModulo512(N, MesLen.Reverse().ToArray());  InterimResults.N.Add(InterimResults.ConvertToBinaryString(N));  Sigma = AddModulo512(Sigma, paddedMes);  h = G\_n(N\_0, h, N);  InterimResults.H.Add(InterimResults.ConvertToBinaryString(h));  h = G\_n(N\_0, h, Sigma);  InterimResults.H.Add(InterimResults.ConvertToBinaryString(h));  if (outLen == 512)  {  InterimResults.ResultHash = BitConverter.ToString(h);  return h;  }  else  {  byte[] h256 = new byte[32];  Array.Copy(h, 0, h256, 0, 32);  return h256;  }  }  }  } |