**ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ**

ПО - программное обеспечение.

ОС – Операционная система.

IP – Internet Protocol - уникальный адрес узла в сети, используемый для маршрутизации пакетов.

Порт - Числовой идентификатор сетевого процесса, применяемый для адресации внутри узла при работе по протоколам TCP/UDP.

UDP - User Datagram Protocol - протокол передачи данных без установления соединения, используемый в сетевых приложениях с низкими задержками.

TAP – Terminal Access Point – виртуальный сетевой интерфейс в Linux.

ECDH - Elliptic Curve Diffie-Hellman - протокол согласования общего ключа.

Curve25519 - Эллиптическая кривая, используемая в ECDH.

AEAD - Authenticated Encryption with Associated Data - режим шифрования с аутентификацией.

ChaCha20 – Симметричный потоковый шифр.

Poly1305 - Алгоритм MAC (Message Authentication Code) для проверки подлинности.

nonce - Одноразовое случайное число, используемое в шифровании.

SHA-256 - Криптографическая хеш-функция с выходом 256 бит.

GUI – Graphical User Interface – графический пользовательский интерфейс.

POSIX - Интерфейс для сетевых операций, определенный стандартом POSIX (Portable Operating System Interface), используемый в Unix-подобных системах.

Winsock - Windows Sockets API, интерфейс для сетевых операций в операционной системе Windows.

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 3](#_Toc195751155)

[1 НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ 4](#_Toc195751156)

[2 АРХИТЕКТУРА СИСТЕМЫ И ПРОЦЕСС ШИФРОВАНИЯ 5](#_Toc195751157)

[2.1 Применяемы технологии и инструменты 5](#_Toc195751158)

[2.2 Общая структура и взаимодействие компонентов 5](#_Toc195751159)

[2.3 Формат и обработка данных 7](#_Toc195751160)

[2.4 Режимы работы и конфигурация 8](#_Toc195751161)

[2.5 Графический интерфейс 9](#_Toc195751162)

[3 КРИПТОГРАФИЧЕСКИЕ АЛГОРИТМЫ И МЕХАНИЗМЫ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ. 11](#_Toc195751163)

[3.1 Алгоритм ChaCha20-Poly1305 (AEAD) 11](#_Toc195751164)

[3.2 Хеш-функция SHA-256 и контроль целостности 12](#_Toc195751165)

[3.3 Генерация случайных значений (ключей и nonce) 12](#_Toc195751166)

[3.4 Автоматический обмен ключами (ECDH Curve25519) 13](#_Toc195751167)

[3.5 Структура зашифрованного пакета 14](#_Toc195751168)

[4 ТЕСТИРОВАНИЕ И ПРОВЕРКА КОРРЕКТНОСТИ РАБОТЫ 16](#_Toc195751169)

[4.1 Проверка корректности расшифровки 16](#_Toc195751170)

[4.2 Среда и сценарии тестирования 16](#_Toc195751171)

[4.3 Результаты измерений производительности 17](#_Toc195751172)

[ЗАКЛЮЧЕНИЕ 18](#_Toc195751173)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 19](#_Toc195751174)

[ПРИЛОЖЕНИЕ А – Исходный код файла tap\_encrypt.cpp 20](#_Toc195751175)

[ПРИЛОЖЕНИЕ Б - Исходный код файла tap\_decrypt.cpp 29](#_Toc195751176)

[ПРИЛОЖЕНИЕ В - Исходный код файла test\_speed.cpp 37](#_Toc195751177)

# ВВЕДЕНИЕ

Настоящий документ описывает разработанное программное обеспечение (ПО) для защиты трафика на канальном уровне при передаче Ethernet-кадров в системе Li-Fi, где данные распространяются посредством световых волн. Система ориентирована на сценарии с географически разнесёнными сегментами локальной вычислительной сети и должна обеспечивать шифрование и аутентификацию передаваемых кадров в соответствии с требованиями нормативных актов Республики Беларусь [1, 2].

ПО реализует полноценный цикл шифрования и дешифрования Ethernet-кадров и текстовых сообщений. Архитектура ПО изначально проектировалась с учётом возможности работы на двух компьютерах: передающая и принимающая стороны запускаются независимо. Для тестирования используется пара виртуальных TAP-интерфейсов (tap0, tap1) на одном компьютере.

Передача зашифрованных данных осуществляется через UDP-протокол, при этом IP-адрес и порт задаются вручную при запуске программ, что позволяет использовать систему не только для локального тестирования, но и в реальной сети между двумя устройствами.

При развёртывании на реальном Li-Fi-оборудовании архитектура сохраняется: данные передаются через UDP, а все механизмы шифрования, дешифрования, обмена ключами и контроля целостности остаются неизменными.

# 1 НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ

Программное обеспечение (ПО) предназначено для защиты трафика на канальном уровне в системе связи Li-Fi, что соответствует требованиям нормативных актов Республики Беларусь в области защиты информации и международным стандартам [1; 2].

Оно обеспечивает шифрование и проверку целостности передаваемых Ethernet-кадров при взаимодействии между географически распределёнными сегментами локальной вычислительной сети. Передача данных осуществляется через световую среду (Li-Fi), а защита информации реализуется на программном уровне с использованием современных криптографических алгоритмов [4; 7].

Программный комплекс включает несколько взаимосвязанных модулей:

- tap\_encrypt.cpp – компонент, отвечающий за захват, шифрование и отправку Ethernet-кадров;

- tap\_decrypt.cpp – компонент, принимающий зашифрованные данные, выполняющий их расшифровку и подачу обратно в сетевой стек.

ПО работает с использованием виртуальных TAP-интерфейсов, которые представляют собой программные сетевые устройства, эмулирующие работу физического Ethernet-интерфейса на канальном уровне модели OSI. Такие интерфейсы позволяют захватывать и передавать Ethernet-кадры напрямую из пользовательского пространства, что позволяет встраивать его в реальные и тестовые среды без необходимости модификации физической топологии сети.

Все кадры, проходящие через TAP-интерфейс, инкапсулируются, шифруются и передаются по каналу связи на основе протокола UDP (User Datagram Protocol), который обеспечивает быструю отправку пакетов без установления соединения, что особенно важно в условиях работы с ограниченными по времени и ресурсоёмкости каналами, такими как Li-Fi.

Разработка ориентирована на следующие облаcти применения:

- защищённые канальные соединения в распределённых вычислительных сетях;

- системы промышленного интернета вещей (IoT);

- интеграция в Li-Fi-инфраструктуру для повышения уровня информационной безопасности;

- образовательные и научно-исследовательские цели в области сетевой безопасности;

- транспортные и мобильные решения, где необходима защита передачи данных между узлами.

Благодаря применению симметричного шифрования (ChaCha20-Poly1305) и автоматического обмена ключами (ECDH Curve25519), ПО обеспечивает конфиденциальность, целостность и аутентичность информации, передаваемой в световом канале связи.

# 2 АРХИТЕКТУРА СИСТЕМЫ И ПРОЦЕСС ШИФРОВАНИЯ

## 2.1 Применяемы технологии и инструменты

Программное обеспечение реализовано на языке программирования C++. Для автоматизации сборки используется система CMake.

Криптографические операции реализованы с использованием библиотеки libsodium [7], поддерживающей ОС Linux и Windows. Применяются алгоритмы ChaCha20-Poly1305 (режим AEAD) [6], SHA-256 [4], а также функции генерации и согласования общего симметричного ключа по протоколу ECDH на эллиптической кривой Curve25519 [5].

Передача данных осуществляется через UDP-сокеты, реализованные с использованием системных функций socket(), sendto(), recvfrom() [3].

Для захвата и подачи Ethernet-кадров используются виртуальные TAP-интерфейсы (/dev/net/tun). Работа с интерфейсами организована через вызовы ioctl, read и write. Для получения всех кадров применяется режим promiscuous mode.

Графический интерфейс реализован с использованием фреймворка Qt и обеспечивает запуск соответствующих модулей, настройку IP-адреса, порта и режима работы (в том числе текстового режима) без обращения к терминалу.

Программное обеспечение разрабатывалось и тестировалось в среде Linux. Адаптация под Windows возможна благодаря кроссплатформенности Libsodium и Qt. Для этого требуется замена взаимодействия с TAP-интерфейсами на драйвер TAP-Windows и переработка сетевых операций с использованием Winsock вместо POSIX-сокетов. После указанных изменений программное обеспечение сохранит функциональность шифрования, обмена ключами и передачи данных. Тестирование в Windows не проводилось, но кроссплатформенная архитектура допускает такую адаптацию.

## 2.2 Общая структура и взаимодействие компонентов

Разработанное программное обеспечение представляет собой систему защищённой передачи данных на канальном уровне, функционирующую поверх световой среды передачи (Li-Fi). Архитектура решения основана на использовании двух независимых модулей:

- tap\_encrypt – компонент, выполняющий захват, шифрование и отправку Ethernet-кадров через виртуальный интерфейс tap0;

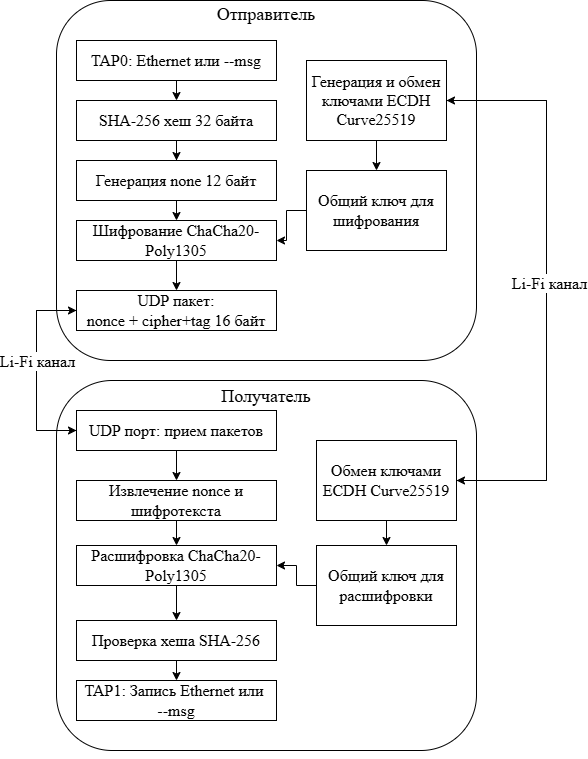
- tap\_decrypt – компонент, принимающий зашифрованные данные, осуществляющий расшифровку и передачу кадров обратно в сетевой стек через tap1.

Взаимодействие между компонентами осуществляется посредством протокола UDP (User Datagram Protocol), не требующего установления соединения, что позволяет минимизировать задержки и упростить маршрутизацию в условиях нестабильных или высокоскоростных сред, таких как Li-Fi.

Оба модуля используют **виртуальные TAP-интерфейсы**, которые эмулируют работу физических Ethernet-устройств на канальном уровне модели OSI. Это позволяет осуществлять захват и ввод кадров без изменения топологии или конфигурации физической сети.

Перед началом передачи данных между отправителем и получателем выполняется **автоматический обмен публичными ключами**, реализованный с помощью протокола ECDH (Elliptic Curve Diffie-Hellman) на эллиптической кривой Curve25519 [5]. В результате обмена обе стороны независимо вычисляют общий симметричный ключ, который используется в алгоритмах шифрования и расшифровки. Этот ключ не передаётся по сети, а вычисляется локально, что обеспечивает криптографическую стойкость канала.

Общая схема взаимодействия компонентов представлена на рисунке 2.1.



**Рисунок 2.1 – Схема взаимодействия компонентов при передаче зашифрованного Ethernet-кадра в системе Li-Fi**

Схема разделена на две логические части: в верхней части отображён процесс обработки и шифрования данных на стороне отправителя, в нижней - этапы приёма, расшифровки и проверки целостности данных на стороне получателя.

На стороне отправителя данные поступают из виртуального сетевого интерфейса tap0, где происходит захват исходных Ethernet-кадров или текстовых сообщений (в зависимости от режима работы). Перед шифрованием от исходного содержимого рассчитывается хеш-функция SHA-256, которая затем объединяется с самими данными. Сформированный открытый блок шифруется с использованием алгоритма AEAD ChaCha20-Poly1305, при этом для каждого пакета генерируется уникальный одноразовый вектор инициализации (nonce). На выходе получается зашифрованный фрагмент, сопровождаемый тегом аутентичности, который вместе с nonce формирует UDP-пакет, отправляемый на приёмную сторону.

На стороне приёмника процесс начинается с приёма UDP-пакета, из которого извлекаются nonce, зашифрованный блок и тег аутентичности. Далее выполняется расшифровка с использованием ранее вычисленного общего ключа. В случае успешной верификации тега расшифрованные данные разбиваются на хеш и полезную нагрузку. Получатель повторно рассчитывает SHA-256 от данных и сравнивает с переданным хешем для проверки целостности. Если проверка пройдена, данные записываются в интерфейс tap1, попадая в сетевой стек системы. Если верификация неудачна, пакет отбрасывается.

## 2.3 Формат и обработка данных

Передача данных между программными модулями осуществляется с применением современных криптографических механизмов. В состав каждого передаваемого блока входят: хеш SHA-256, исходный Ethernet-кадр, уникальный одноразовый вектор инициализации (nonce), а также тег аутентификации.

Перед шифрованием от исходного кадра рассчитывается хеш-функция SHA-256, размером 32 байта. Полученное значение объединяется с самим кадром, образуя открытые данные для шифрования. Далее формируется nonce размером 12 байт, генерируемый случайным образом для каждого отдельного пакета. Шифрование осуществляется с использованием алгоритма AEAD ChaCha20-Poly1305, который одновременно обеспечивает конфиденциальность и аутентификацию данных. В результате формируется зашифрованный блок и тег аутентичности размером 16 байт.

Итоговая структура UDP-пакета имеет следующий вид:

*[12 байт nonce][зашифрованные данные][16 байт тега аутентичности]*

Зашифрованная область включает в себя как хеш SHA-256, так и сам Ethernet-кадр. Такая конструкция позволяет осуществлять проверку целостности уже после расшифровки.

На стороне отправителя (tap\_encrypt) данные считываются из интерфейса tap0, обрабатываются (хешируются, объединяются), шифруются и направляются через UDP-сокет на заранее указанный IP-адрес и порт.

На стороне приёмника (tap\_decrypt) производится выделение nonce и зашифрованной части пакета, расшифровка с верификацией тега Poly1305, последующее повторное вычисление хеша и его сравнение с переданным значением. При совпадении расшифрованные данные направляются в интерфейс tap1 и становятся доступны в сетевом стеке операционной системы. В случае обнаружения несоответствия данных пакет считается повреждённым и отбрасывается.

Представленный формат передачи обеспечивает одновременно как конфиденциальность, так и контроль целостности передаваемой информации, что критически важно для систем передачи по световому каналу (Li-Fi), особенно в условиях возможных потерь и помех.

## 2.4 Режимы работы и конфигурация

Разработанное программное обеспечение поддерживает два режима функционирования:

- режим передачи Ethernet-кадров (основной);

- режим обмена текстовыми сообщениями, активируемый посредством аргумента командной строки --msg.

В режиме по умолчанию (tap\_encrypt и tap\_decrypt без дополнительных флагов) осуществляется захват, шифрование, передача и расшифровка реальных Ethernet-кадров через TAP-интерфейсы tap0 и tap1 соответственно. Этот режим предназначен для использования в реальных сетевых условиях и позволяет прозрачно шифровать любой трафик, проходящий через виртуальный интерфейс.

Режим --msg представляет собой специальный текстовый режим, предназначенный для ручного ввода и передачи сообщений. Пользователь может вводить строки в терминале, после чего они будут зашифрованы на стороне отправителя и отображены в расшифрованном виде на стороне приёмника. Этот режим полезен для тестирования, демонстрации работы алгоритма шифрования и отладки программного взаимодействия.

Конфигурация IP-адреса и порта для взаимодействия осуществляется через аргументы командной строки. Вызов программ может быть выполнен следующим образом:

*./tap\_encrypt [ip] [port]*

*./tap\_decrypt [ip] [port]*

Если параметры не заданы, используются значения по умолчанию:

* для tap\_encrypt: IP-адрес 127.0.0.1, порт 12345;
* для tap\_decrypt: IP-адрес 0.0.0.0, порт 12345.

Такой подход обеспечивает гибкость и адаптацию к различным сценариям применения: как при локальном запуске на одном компьютере, так и при взаимодействии между двумя физически разнесёнными устройствами по реальной сети или Li-Fi-каналу.

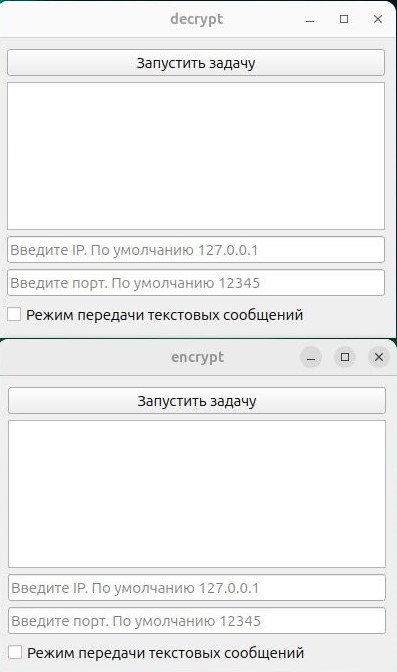
Программа также поддерживает работу в изолированных сетевых пространствах (net namespaces), что позволяет выполнять тестирование в условиях, приближённых к многомашинной среде. Поддерживаются как локальные сценарии, так и запуск через виртуальные интерфейсы внутри отдельных сетевых пространств с использованием инструментов ip netns, tcpdump, hping3, iperf и других утилит для генерации трафика и оценки пропускной способности.

Таким образом, реализация программного обеспечения учитывает различные режимы работы, обеспечивает ручную и автоматическую конфигурацию параметров, а также позволяет гибко адаптировать систему под цели тестирования, демонстрации или практического использования в составе защищённой канальной подсистемы на базе Li-Fi.

## 2.5 Графический интерфейс

Для повышения удобства взаимодействия с программным обеспечением был разработан графический пользовательский интерфейс (GUI). Интерфейс предоставляет пользователю возможность конфигурировать параметры соединения и выбирать режим работы без необходимости обращения к командной строке.

Графическая оболочка состоит из двух окон: одно предназначено для запуска модуля отправки данных (tap\_encrypt), второе — для приёма и расшифровки (tap\_decrypt). В каждом окне предусмотрены элементы управления, позволяющие задать IP-адрес, порт подключения и активировать режим передачи текстовых сообщений. Также реализована кнопка запуска задачи и окно логов, в котором отображаются сообщения о ходе выполнения процесса: успешные соединения, полученные или отправленные пакеты, ошибки верификации и другие диагностические сведения. На рисунке 2.2 представлен внешний вид пользовательского интерфейса.



**Рисунок 2.2 – Графический интерфейс модулей передачи и приема зашифрованных кадров.**

Наличие графического интерфейса упрощает тестирование, демонстрацию работы системы и настройку соединений. Он делает программное обеспечение доступным для широкой аудитории, включая пользователей, не имеющих опыта работы в терминале. GUI также может использоваться в демонстрационных целях, где важна наглядность и скорость конфигурации среды.

# 3 КРИПТОГРАФИЧЕСКИЕ АЛГОРИТМЫ И МЕХАНИЗМЫ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ

## 3.1 Алгоритм ChaCha20-Poly1305 (AEAD)

ChaCha20-Poly1305 представляет собой комбинацию потокового шифра ChaCha20 и алгоритма аутентификации Poly1305, реализующую режим AEAD (Authenticated Encryption with Associated Data) [6], обеспечивающий конфиденциальность и аутентификацию данных [4].

Режим AEAD представляет собой способ симметричного шифрования, при котором одновременно обеспечиваются конфиденциальность данных и их аутентификация. Он включает генерацию тега подлинности, позволяющего выявить любые модификации зашифрованного сообщения.

ChaCha20 - это потоковый шифр, генерирующий псевдослучайную последовательность (ключевой поток) длиной, соответствующей размеру входных данных. Шифрование осуществляется побитовой операцией XOR (исключающее «ИЛИ») между ключевым потоком и открытым текстом.

Poly1305 - это алгоритм вычисления кода аутентификации сообщения (MAC, Message Authentication Code), используемый в составе схемы ChaCha20-Poly1305. Он обеспечивает проверку подлинности зашифрованных данных и защиты от их модификации.

В отличие от хеш-функций, Poly1305 не является необратимой функцией, а работает с заранее известным секретным ключом и предназначен именно для контроля целостности и подтверждения подлинности сообщения.

Poly1305 принимает на вход зашифрованный текст и дополнительный внутренний ключ, сгенерированный на основе основного симметричного ключа и nonce. Результатом вычисления является 16-байтный тег аутентификации (auth tag), который передаётся вместе с данными и nonce в составе UDP-пакета.

Алгоритм работает по следующей схеме:

1. Генерируется одноразовое значение nonce (12 байт).
2. Формируется ключевой поток на основе ChaCha20, используя ключ и nonce.
3. Исходные данные шифруются с помощью XOR с ключевым потоком.
4. Тег Poly1305 вычисляется по зашифрованному сообщению.
5. При передаче используется структура пакета: [nonce][ciphertext][auth\_tag]

Параметры алгоритма:

- длина ключа - 256 бит;

- длина nonce - 96 бит;

- длина тега - 128 бит.

Режим AEAD означает, что шифруемые данные защищены не только от чтения, но и от модификации. Любое изменение шифротекста или тега приводит к невозможности расшифровки. Это обеспечивает как конфиденциальность, так и целостность данных.

В проекте алгоритм реализуется через функции библиотеки libsodium. Генерация nonce выполняется случайным образом для каждого пакета. Повторное использование одного и того же nonce с одинаковым ключом не допускается.

## 3.2 Хеш-функция SHA-256 и контроль целостности

Для контроля целостности передаваемых данных в проекте используется криптографическая хеш-функция SHA-256 [4]. Хеш-функция представляет собой одностороннее преобразование, формирующее из произвольного входа фиксированный 256-битный выход. Перед шифрованием каждый Ethernet-кадр или текстовое сообщение проходит через процедуру хеширования, в результате которой формируется 32-байтное значение - хеш-сумма.

В полученном зашифрованном сообщении первым блоком размещается результат хеширования (SHA-256) от исходных данных, за которым следует сам исходный блок. Общая структура открытых данных до шифрования: [hash || payload], где hash - результат SHA-256, payload - исходное содержимое.

После расшифровки, на стороне приёмника, выполняется повторное хеширование полученного открытого блока. Вычисленный хеш сравнивается с тем, что был передан. В случае совпадения данные считаются корректными. При расхождении хешей пакет отбрасывается.

Применение SHA-256 позволяет обнаруживать искажения данных, возникшие как в результате ошибок передачи, так и при попытках злонамеренной модификации. Использование данной функции до шифрования обеспечивает дополнительный уровень защиты, независимый от встроенного тега аутентичности в ChaCha20-Poly1305.

Хеш-функция SHA-256 реализуется средствами библиотеки libsodium. Её стойкость основана на невозможности восстановления исходных данных по хешу, а также на высокой чувствительности к изменению даже одного бита входного сообщения.

## 3.3 Генерация случайных значений (ключей и nonce)

В проекте для генерации симметричных ключей и одноразовых значений (nonce) используется встроенный в библиотеку libsodium криптографически стойкий генератор псевдослучайных чисел randombytes\_buf() [7].

Функция randombytes\_buf() обеспечивает генерацию последовательностей байтов, обладающих высокой энтропией и устойчивостью к предсказанию. Для генерации ключей используется размер 256 бит (32 байта), для nonce - 96 бит (12 байт). Эти значения соответствуют требованиям алгоритма ChaCha20-Poly1305.

Генерация ключевой пары для последующего согласования общего симметричного ключа (ECDH Curve25519) также осуществляется с использованием функций crypto\_kx\_keypair(), предоставляемых библиотекой libsodium. Приватные ключи не передаются по сети и используются только локально при расчётах.

Nonce генерируется независимо для каждого передаваемого пакета. Повторное использование одного и того же nonce с тем же ключом не допускается, так как это приводит к нарушению криптографической стойкости алгоритма AEAD. Уникальность nonce обеспечивает независимость шифротекстов даже при совпадении открытых данных.

Использование криптографически стойкого генератора случайных чисел является обязательным требованием при реализации защищённых каналов связи. В рамках проекта генерация случайных значений полностью соответствует требованиям, обеспечивая безопасность на этапе подготовки каждого сообщения к отправке.

## 3.4 Автоматический обмен ключами (ECDH Curve25519)

В проекте используется протокол согласования симметричного ключа на основе эллиптической криптографии - **ECDH (Elliptic Curve Diffie-Hellman),** реализованный на кривой Curve25519 [5]**.** Он позволяет двум сторонам безопасно сформировать общий секрет, не передавая его по сети. Алгоритм применяется для установления ключа, используемого в дальнейшем в симметричном шифровании (ChaCha20-Poly1305).

Протокол ECDH является развитием классического алгоритма Диффи-Хеллмана и использует арифметику над точками на эллиптической кривой. В проекте применяется **кривая Curve25519**, рекомендованная для практического применения в современных криптографических системах. Она определяет специальную группу точек с характеристиками, обеспечивающими стойкость к известным типам атак, а также эффективные операции умножения точки на скаляр.

Работа ECDH включает следующие этапы:

1. **Генерация ключевой пары.** Каждая сторона (отправитель и получатель) генерирует свою пару ключей: закрытый (приватный) и открытый (публичный). Приватный ключ представляет собой случайное число, а открытый вычисляется как произведение базовой точки кривой на приватный ключ.
2. **Обмен публичными ключами.** Стороны передают друг другу только публичные ключи. Приватные ключи остаются на стороне генерации и участвуют только в локальных вычислениях.
3. **Формирование общего секрета.** Каждая сторона умножает полученный чужой публичный ключ на свой приватный. В результате обе стороны получают одинаковый элемент группы - **общий секрет**, из которого может быть получен симметричный ключ.

Генерация ключей и вычисление общего секрета выполняются с использованием функций библиотеки libsodium.

Для генерации пары ключей применяется функция crypto\_kx\_keypair().  
Расчёт общего симметричного ключа выполняется с помощью:

* crypto\_kx\_client\_session\_keys() - на стороне отправителя;
* crypto\_kx\_server\_session\_keys() - на стороне получателя.

Поскольку расчёт производится с использованием различных ролей (client/server), итоговые симметричные ключи совпадают.

Обмен осуществляется один раз при запуске программ, до начала основной передачи данных. Вычисленный общий ключ используется при шифровании и расшифровке пакетов по алгоритму ChaCha20-Poly1305.

Преимущество ECDH на Curve25519 заключается в высокой криптографической стойкости, скорости выполнения и отсутствии необходимости в предварительном согласовании ключей через внешние каналы. Такой подход обеспечивает безопасное установление защищённого канала без хранения или ручной передачи ключей.

## 3.5 Структура зашифрованного пакета

Каждое передаваемое сообщение инкапсулируется в UDP-пакет, содержащий три основных элемента: одноразовое значение nonce, зашифрованные данные и тег аутентификации. Такая структура используется для всех типов передаваемой информации (кадры или текстовые сообщения) и соответствует требованиям алгоритма ChaCha20-Poly1305 в режиме AEAD.

Формат пакета имеет следующий вид:

[nonce (12 байт)][ciphertext][auth\_tag (16 байт)]

Содержимое ciphertext представляет собой результат шифрования открытых данных, включающих:

- хеш SHA-256 от исходного содержимого (32 байта);

- исходные данные (кадр Ethernet или текстовое сообщение).

Таким образом, фактически шифруется блок следующей структуры:

[hash || payload], где hash – результат SHA256, payload – исходное сообщение.

Nonce представляет собой уникальное случайное значение, генерируемое для каждого пакета. Его использование исключает повторное шифрование одинаковых данных с тем же ключом и обеспечивает криптографическую стойкость.

Тег аутентификации (auth\_tag) генерируется с помощью алгоритма Poly1305 и позволяет проверить целостность и подлинность сообщения. В случае несоответствия тега расшифровка не выполняется, а пакет отбрасывается.

Такая структура позволяет передавать защищённые данные без установки соединения, с минимальными накладными расходами и полной поддержкой контроля целостности и конфиденциальности.

Преимущество предложенной структуры заключается в возможности последующего расширения без нарушения совместимости. При необходимости к формату пакета могут быть добавлены дополнительные поля, такие как идентификатор сессии или временные метки, без ущерба для основного алгоритма обработки.

# 4 ТЕСТИРОВАНИЕ И ПРОВЕРКА КОРРЕКТНОСТИ РАБОТЫ

## 4.1 Проверка корректности расшифровки

Проверка корректности данных осуществляется встроенным механизмом сравнения хешей, реализованным внутри программы на обоих этапах - шифрования и расшифровки. Хеш SHA-256 от каждого исходного кадра или сообщения добавляется в начало блока перед шифрованием.

После расшифровки на стороне получателя первые 32 байта интерпретируются как «присланный хеш». Остальная часть сообщения повторно хешируется. Далее выполняется сравнение полученного значения с присланным. Если хеши совпадают, выводится сообщение о корректности («Хеши совпадают - кадр корректен»), и данные передаются в TAP-интерфейс. При расхождении хешей кадр считается повреждённым и отбрасывается.

Для наглядности включён вывод результатов проверки непосредственно в консоль, как в режиме передачи кадров, так и в текстовом режиме (--msg). Механизм идентичен для обеих реализаций и не зависит от типа данных, передаваемых через защищённый канал.

## 4.2 Среда и сценарии тестирования

Тестирование программного обеспечения проводилось в двух конфигурациях: в локальной среде на одном устройстве и в изолированной сетевой инфраструктуре с использованием пространств имён (netns). Целью испытаний являлась проверка стабильности, корректности работы алгоритмов шифрования и расшифровки, а также устойчивости системы в различных режимах запуска.

В локальном сценарии оба модуля - tap\_encrypt и tap\_decrypt - запускались на одном компьютере. Для симуляции передачи использовались виртуальные сетевые интерфейсы tap0 и tap1, а взаимодействие осуществлялось по протоколу UDP через адрес 127.0.0.1 и порт 12345. Такой подход позволил убедиться в базовой работоспособности системы, корректности обработки кадров и функционировании встроенной проверки целостности данных.

Для эмуляции взаимодействия между двумя независимыми устройствами применялись сетевые неймспейсы ns1 и ns2, между которыми настраивались виртуальные интерфейсы с IP-адресацией. Внутри каждого пространства имён запускались соответствующие компоненты системы, а передача данных осуществлялась через зашифрованный UDP-канал. Такой сценарий позволил протестировать работу системы при пространственном разнесении модулей и оценить устойчивость при реальном сетевом обмене.

Во всех сценариях система демонстрировала стабильную работу, корректную обработку кадров и сообщений, и корректное реагирование на ошибки целостности. Тестирование подтвердило возможность гибкого развёртывания решения как в локальной, так и в распределённой среде без изменения внутренней логики шифрования и дешифрования.

## 4.3 Результаты измерений производительности

Оценка производительности ПО проводилась по двум направлениям: скорость выполнения криптографических операций и фактическая пропускная способность защищённого канала связи при передаче данных через TAP-интерфейсы по UDP-протоколу.

Для оценки скорости шифрования и расшифровки использовался модуль test\_speed.cpp, обрабатывающий блок данных объёмом 10 МБ с применением алгоритма ChaCha20-Poly1305.Время выполнения операции измерялось с помощью стандартных средств языка C++, после чего вычислялась средняя пропускная способность. Полученные значения стабильно превышали 10 МБ/с, что соответствует целевым требованиям проекта и позволяет использовать систему в средах с высокими требованиями к скорости обработки данных, включая каналы Li-Fi и беспроводные IoT-сети.

Дополнительно проводились интеграционные испытания в сетевой конфигурации с использованием TAP-интерфейсов и UDP. Передача трафика осуществлялась от одного виртуального интерфейса к другому с полным циклом обработки: захват, хеширование, шифрование, передача, расшифровка, проверка целостности и возврат в стек.

Для оценки параметров канала применялись стандартные утилиты ping и iperf. С их помощью проверялась доступность шифрованного соединения, наличие потерь, а также измерялась фактическая пропускная способность защищённого канала. По результатам тестирования было установлено, что система обеспечивает стабильную передачу данных без потерь и с пропускной способностью, значительно превышающей 10 Мбит/с, что полностью удовлетворяет требованиям, изложенным в техническом задании.

Полученные результаты подтверждают, что система устойчива при нагрузке и обладает запасом по производительности. Реализация поддерживает как одностороннюю, так и двунаправленную передачу, что позволяет использовать её в различных конфигурациях без изменения логики обработки.

# ЗАКЛЮЧЕНИЕ

В рамках проекта было разработано программное обеспечение для защиты трафика на канальном уровне с применением современных криптографических алгоритмов. Реализация включает модули шифрования и расшифровки, поддержку TAP-интерфейсов, автоматический обмен ключами по протоколу ECDH, а также графический интерфейс для конфигурации параметров [5; 7].

Система обеспечивает двунаправленную передачу данных между двумя узлами, реализуя полный цикл шифрования, расшифровки и проверки целостности передаваемых кадров. Тестирование подтвердило стабильную работу в различных режимах, соответствие заявленным требованиям и достижение пропускной способности выше 10 Мбит/с.

Архитектура решения допускает масштабирование, расширение набора поддерживаемых протоколов и возможность интеграции в аппаратные платформы. Полученные результаты подтверждают возможность практического применения системы в условиях реального времени и повышенных требований к безопасности.

Проведённая реализация охватывает полный цикл обработки Ethernet-кадров, включая формирование хеша, шифрование, передачу, расшифровку и контроль целостности. Структура системы построена с учётом изоляции логики приёма и отправки, что упрощает сопровождение и возможную интеграцию с внешними модулями. Программный комплекс может быть адаптирован для использования в составе специализированных решений в области связи, информационной безопасности и встроенных систем.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Приказ Оперативно-аналитического центра при Президенте Республики Беларусь от 20 февраля 2020 г. № 66 «О мерах по реализации Указа Президента Республики Беларусь от 9 декабря 2019 г. № 449» Оперативно-аналитический центр при Президенте Республики Беларусь [Электронный ресурс]. Режим доступа: - <https://www.oac.gov.by/public/content/files/files/law/prikaz-oac/2020%20-%2066.pdf>. - Дата доступа : 16.04.2025.
2. Технический регламент Республики Беларусь ТР 2013/027/BY «Информационные технологии. Средства защиты информации. Информационная безопасность» : утв. постановлением Совета Министров Республики Беларусь от 15 мая 2013 г. № 375. Оперативно-аналитический центр при Президенте Республики Беларусь [Электронный ресурс] - Режим доступа: - <https://www.oac.gov.by/public/content/files/files/law/resolutions-sm/2013%20-%20375.pdf>. - Дата доступа: 16.04.2025.
3. Таненбаум, Э. Компьютерные сети : пер. с англ. / Э. Таненбаум, Д. Уэзеролл. - 5-е изд. - СПб. : Питер, 2012. - 960 с.
4. Шнайер, Б. Прикладная криптография : протоколы, алгоритмы и исходный код на языке С : пер. с англ. / Б. Шнайер. - М. : Триумф, 2002. - 816 с.
5. Bernstein, D. J. Curve25519 : New Diffie-Hellman Speed Records / D. J. Bernstein // Public Key Cryptography - PKC 2006 : 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006 : proceedings / ed. M. Yung [et al.]. - Berlin : Springer, 2006. - P. 207-228. - Режим доступа : https://cr.yp.to/ecdh/curve25519-20060209.pdf. - Дата доступа : 12.04.2025.
6. Nir, Y. ChaCha20 and Poly1305 for IETF Protocols : RFC 8439 / Y. Nir, A. Langley [Электронный ресурс]. - Internet Engineering Task Force (IETF), 2018. - Режим доступа : https://datatracker.ietf.org/doc/html/rfc8439. - Дата доступа : 12.04.2025.
7. Libsodium Documentation [Электронный ресурс]. - Режим доступа : https://libsodium.gitbook.io/doc/. - Дата доступа : 15.04.2025.

# ПРИЛОЖЕНИЕ А – Исходный код файла tap\_encrypt.cpp

#include <iostream>

#include <fstream>

#include <vector>

#include <cstring>

#include <fcntl.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <linux/if\_tun.h>

#include <netinet/in.h>

#include <net/if.h>

#include <sys/socket.h>

#include <sodium.h>

#include <arpa/inet.h> // для inet\_pton

#include <iomanip>     // для std::setw, std::setfill

#include <thread>

constexpr size\_t MAX\_PACKET\_SIZE = 2000;

constexpr size\_t KEY\_SIZE = crypto\_aead\_chacha20poly1305\_IETF\_KEYBYTES;

constexpr size\_t NONCE\_SIZE = crypto\_aead\_chacha20poly1305\_IETF\_NPUBBYTES;

constexpr size\_t HASH\_SIZE = crypto\_hash\_sha256\_BYTES;

int open\_tap(const std::string &dev\_name)

{

    struct ifreq ifr{};

    int fd = open("/dev/net/tun", O\_RDWR);

    if (fd < 0)

    {

        perror("open /dev/net/tun");

        exit(1);

    }

    ifr.ifr\_flags = IFF\_TAP | IFF\_NO\_PI;

    std::strncpy(ifr.ifr\_name, dev\_name.c\_str(), IFNAMSIZ);

    if (ioctl(fd, TUNSETIFF, &ifr) < 0)

    {

        perror("ioctl TUSETIFF");

        close(fd);

        exit(1);

    }

    return fd;

}

void receive\_frames(int tap\_fd, int sock, const std::vector<unsigned char> &key)

{

    while (true)

    {

        unsigned char buffer[MAX\_PACKET\_SIZE];

        ssize\_t nrecv = recv(sock, buffer, sizeof(buffer), 0);

        if (nrecv <= NONCE\_SIZE)

            continue;

        std::vector<unsigned char> nonce(buffer, buffer + NONCE\_SIZE);

        std::vector<unsigned char> ciphertext(buffer + NONCE\_SIZE, buffer + nrecv);

        std::vector<unsigned char> decrypted(ciphertext.size());

        unsigned long long decrypted\_len = 0;

        int result = crypto\_aead\_chacha20poly1305\_ietf\_decrypt(

            decrypted.data(), &decrypted\_len,

            nullptr,

            ciphertext.data(), ciphertext.size(),

            nullptr, 0,

            nonce.data(), key.data());

        if (result != 0)

        {

            std::cerr << " Ошибка расшифровки в receive\_frames!\n";

            continue;

        }

        if (decrypted\_len < HASH\_SIZE)

        {

            std::cerr << " Слишком маленький расшифрованный буфер!\n";

            continue;

        }

        unsigned char received\_hash[HASH\_SIZE];

        std::memcpy(received\_hash, decrypted.data(), HASH\_SIZE);

        size\_t msg\_len = decrypted\_len - HASH\_SIZE;

        unsigned char actual\_hash[HASH\_SIZE];

        crypto\_hash\_sha256(actual\_hash, decrypted.data() + HASH\_SIZE, msg\_len);

        if (std::memcmp(received\_hash, actual\_hash, HASH\_SIZE) != 0)

        {

            std::cerr << " Хеш не совпадает в receive\_frames — данные повреждены!\n";

            continue;

        }

        size\_t data\_len = decrypted\_len - HASH\_SIZE;

        std::vector<unsigned char> data\_buf(data\_len);

        std::memcpy(data\_buf.data(), decrypted.data() + HASH\_SIZE, data\_len);

        write(tap\_fd, data\_buf.data(), data\_len);

        std::cout << " Принят и расшифрован кадр из tap1 (" << data\_len << " байт)\n";

    }

}

int main(int argc, char \*argv[])

{

    bool message\_mode = false;

    if (argc >= 2 && std::string(argv[1]) == "--msg")

    {

        message\_mode = true;

        argv++;

        argc--;

    }

    if (sodium\_init() < 0)

    {

        std::cerr << "Не удалось инициализировать libsodium\n";

        return 1;

    }

    const char \*ip\_str = "127.0.0.1";

    int port = 12345;

    if (argc >= 2)

        ip\_str = argv[1];

    if (argc >= 3)

        port = std::stoi(argv[2]);

    std::cout << " Используем IP: " << ip\_str << ", порт: " << port << "\n";

    std::string ping\_cmd = "ping -c 1 " + std::string(ip\_str) + " > /dev/null 2>&1";

    int ping\_result = system(ping\_cmd.c\_str());

    if (ping\_result != 0)

    {

        std::cout << "  Внимание: IP-адрес " << ip\_str

                  << " недоступен (ping не прошёл), но продолжаем...\n";

    }

    else

    {

        std::cout << " IP-адрес " << ip\_str << " доступен, начинаем работу...\n";

    }

    // if (ping\_result != 0)

    // {

    //     std::cout << "  Внимание: IP-адрес " << ip\_str << " недоступен (ping не прошёл)\n";

    //     std::cout << "Продолжить отправку данных? [y/N]: ";

    //     std::string answer;

    //     std::getline(std::cin, answer);

    //     if (answer != "y" && answer != "Y")

    //     {

    //         std::cout << " Отправка отменена пользователем.\n";

    //         return 1;

    //     }

    // }

    // else

    // {

    //     std::cout << " IP-адрес " << ip\_str << " доступен, начинаем работу...\n";

    // }

    // Открываем tap0

    int tap\_fd = open\_tap("tap0");

    std::cout << "📡 tap0 открыт для чтения Ethernet-кадров\n";

    // Создаём UDP-сокет

    int sock = socket(AF\_INET, SOCK\_DGRAM, 0);

    if (sock < 0)

    {

        perror("socket");

        return 1;

    }

    // Формируем адрес назначения

    sockaddr\_in dest\_addr{};

    dest\_addr.sin\_family = AF\_INET;

    dest\_addr.sin\_port = htons(port);

    if (inet\_pton(AF\_INET, ip\_str, &dest\_addr.sin\_addr) <= 0)

    {

        std::cerr << " Неверный IP-адрес\n";

        return 1;

    }

    // === [Автоматический обмен ключами через UDP] ===

    unsigned char my\_public\_key[crypto\_kx\_PUBLICKEYBYTES];

    unsigned char my\_private\_key[crypto\_kx\_SECRETKEYBYTES];

    crypto\_kx\_keypair(my\_public\_key, my\_private\_key);

    // 1. Отправляем свой публичный ключ получателю

    sendto(sock, my\_public\_key, crypto\_kx\_PUBLICKEYBYTES, 0,

           (sockaddr \*)&dest\_addr, sizeof(dest\_addr));

    std::cout << " Публичный ключ отправлен получателю\n";

    // 2. Принимаем публичный ключ от получателя

    unsigned char receiver\_public\_key[crypto\_kx\_PUBLICKEYBYTES];

    ssize\_t received = recv(sock, receiver\_public\_key, crypto\_kx\_PUBLICKEYBYTES, 0);

    if (received != crypto\_kx\_PUBLICKEYBYTES)

    {

        std::cerr << " Ошибка при получении публичного ключа получателя\n";

        return 1;

    }

    std::cout << " Публичный ключ получен от получателя\n";

    // 3. Вычисляем общий ключ (tx\_key)

    std::vector<unsigned char> key(KEY\_SIZE);

    if (crypto\_kx\_client\_session\_keys(nullptr, key.data(),

                                      my\_public\_key, my\_private\_key,

                                      receiver\_public\_key) != 0)

    {

        std::cerr << " Ошибка при расчёте общего ключа (client)\n";

        return 1;

    }

    // Запускаем приём кадров в отдельном потоке

    std::thread receive\_thread(receive\_frames, tap\_fd, sock, std::ref(key));

    // Вектор для nonce (уникальный для каждого кадра)

    std::vector<unsigned char> nonce(NONCE\_SIZE);

    if (message\_mode)

    {

        // Режим текстовых сообщений

        std::cout << " Режим отправки сообщений. Вводите текст:\n";

        std::string user\_message;

        while (std::getline(std::cin, user\_message))

        {

            if (user\_message.empty())

                continue;

            // Считаем SHA-256 от текста

            unsigned char hash\_buf[HASH\_SIZE];

            crypto\_hash\_sha256(hash\_buf,

                               reinterpret\_cast<const unsigned char \*>(user\_message.data()),

                               user\_message.size());

            // Сформируем plaintext = [32 байта хеша] + [исходный текст]

            std::vector<unsigned char> plaintext;

            plaintext.insert(plaintext.end(), hash\_buf, hash\_buf + HASH\_SIZE);

            plaintext.insert(plaintext.end(),

                             reinterpret\_cast<const unsigned char \*>(user\_message.data()),

                             reinterpret\_cast<const unsigned char \*>(user\_message.data()) + user\_message.size());

            // Генерируем nonce

            randombytes\_buf(nonce.data(), nonce.size());

            // Реальный размер plaintext — это (32 + длина сообщения)

            std::vector<unsigned char> encrypted(plaintext.size() + crypto\_aead\_chacha20poly1305\_IETF\_ABYTES);

            unsigned long long encrypted\_len = 0;

            // Шифруем (ChaCha20-Poly1305)

            crypto\_aead\_chacha20poly1305\_ietf\_encrypt(

                encrypted.data(), &encrypted\_len,

                plaintext.data(), plaintext.size(), // <-- здесь

                nullptr, 0, nullptr,

                nonce.data(), key.data());

            // Готовим пакет = nonce + ciphertext

            std::vector<unsigned char> packet;

            packet.insert(packet.end(), nonce.begin(), nonce.end());

            packet.insert(packet.end(), encrypted.begin(), encrypted.begin() + encrypted\_len);

            // Отправляем

            sendto(sock, packet.data(), packet.size(), 0, (sockaddr \*)&dest\_addr, sizeof(dest\_addr));

            std::cout << " Сообщение отправлено (" << user\_message.size() << " байт)\n";

        }

    }

    else

    {

        // Режим отправки Ethernet-кадров из tap

        while (true)

        {

            // Читаем кадр из tap0

            unsigned char buffer[MAX\_PACKET\_SIZE];

            ssize\_t nread = read(tap\_fd, buffer, sizeof(buffer));

            if (nread <= 0)

                continue;

            // Считаем SHA-256 от кадра

            unsigned char hash\_buf[HASH\_SIZE];

            crypto\_hash\_sha256(hash\_buf, buffer, nread);

            // Формируем plaintext = [32 байта хеша] + [сам кадр]

            std::vector<unsigned char> plaintext;

            plaintext.insert(plaintext.end(), hash\_buf, hash\_buf + HASH\_SIZE);

            plaintext.insert(plaintext.end(), buffer, buffer + nread);

            // Генерируем nonce

            randombytes\_buf(nonce.data(), nonce.size());

            // Нужно шифровать plaintext

            std::vector<unsigned char> encrypted(plaintext.size() + crypto\_aead\_chacha20poly1305\_IETF\_ABYTES);

            unsigned long long encrypted\_len = 0;

            crypto\_aead\_chacha20poly1305\_ietf\_encrypt(

                encrypted.data(), &encrypted\_len,

                plaintext.data(), plaintext.size(), // <-- передаём всё

                nullptr, 0, nullptr,

                nonce.data(), key.data());

            // nonce + encrypted

            std::vector<unsigned char> packet;

            packet.insert(packet.end(), nonce.begin(), nonce.end());

            packet.insert(packet.end(), encrypted.begin(), encrypted.begin() + encrypted\_len);

            sendto(sock, packet.data(), packet.size(), 0, (sockaddr \*)&dest\_addr, sizeof(dest\_addr));

            std::cout << " Отправлен зашифрованный кадр (" << nread << " байт)\n";

        }

    }

    close(tap\_fd);

    close(sock);

    return 0;

}

# ПРИЛОЖЕНИЕ Б - Исходный код файла tap\_decrypt.cpp

#include <iostream>

#include <fstream>

#include <vector>

#include <cstring>

#include <fcntl.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <linux/if\_tun.h>

#include <netinet/in.h>

#include <net/if.h>

#include <sys/socket.h>

#include <sodium.h>

#include <arpa/inet.h> // для inet\_pton

#include <thread>

#include <chrono>

#include <sstream>

#include <iomanip>

constexpr size\_t MAX\_PACKET\_SIZE = 2000;

constexpr size\_t KEY\_SIZE = crypto\_aead\_chacha20poly1305\_IETF\_KEYBYTES;

constexpr size\_t NONCE\_SIZE = crypto\_aead\_chacha20poly1305\_IETF\_NPUBBYTES;

constexpr size\_t HASH\_SIZE = crypto\_hash\_sha256\_BYTES;

int open\_tap(const std::string &dev\_name)

{

struct ifreq ifr{};

int fd = open("/dev/net/tun", O\_RDWR);

if (fd < 0)

{

perror("open /dev/net/tun");

exit(1);

}

ifr.ifr\_flags = IFF\_TAP | IFF\_NO\_PI;

std::strncpy(ifr.ifr\_name, dev\_name.c\_str(), IFNAMSIZ);

if (ioctl(fd, TUNSETIFF, &ifr) < 0)

{

perror("ioctl TUNSETIFF");

close(fd);

exit(1);

}

return fd;

}

void send\_frames(int tap\_fd, int sock, const sockaddr\_in &dest\_addr, const std::vector<unsigned char> &key)

{

std::vector<unsigned char> nonce(NONCE\_SIZE);

while (true)

{

unsigned char buffer[MAX\_PACKET\_SIZE];

ssize\_t nread = read(tap\_fd, buffer, sizeof(buffer));

if (nread <= 0)

continue;

unsigned char hash\_buf[HASH\_SIZE];

crypto\_hash\_sha256(hash\_buf, buffer, nread);

std::vector<unsigned char> plaintext;

plaintext.insert(plaintext.end(), hash\_buf, hash\_buf + HASH\_SIZE);

plaintext.insert(plaintext.end(), buffer, buffer + nread);

randombytes\_buf(nonce.data(), nonce.size());

std::vector<unsigned char> encrypted(plaintext.size() + crypto\_aead\_chacha20poly1305\_IETF\_ABYTES);

unsigned long long encrypted\_len = 0;

crypto\_aead\_chacha20poly1305\_ietf\_encrypt(

encrypted.data(), &encrypted\_len,

plaintext.data(), plaintext.size(),

nullptr, 0, nullptr,

nonce.data(), key.data());

std::vector<unsigned char> packet;

packet.insert(packet.end(), nonce.begin(), nonce.end());

packet.insert(packet.end(), encrypted.begin(), encrypted.begin() + encrypted\_len);

sendto(sock, packet.data(), packet.size(), 0, (sockaddr \*)&dest\_addr, sizeof(dest\_addr));

std::cout << " Отправлен зашифрованный кадр из tap1 (" << nread << " байт)\n";

}

}

int main(int argc, char \*argv[])

{

// --msg: если true, тогда мы интерпретируем расшифрованные данные как строку

bool message\_mode = false;

if (argc >= 2 && std::string(argv[1]) == "--msg")

{

message\_mode = true;

argv++;

argc--;

}

if (sodium\_init() < 0)

{

std::cerr << "Не удалось инициализировать libsodium\n";

return 1;

}

// Параметры: IP и порт, на котором слушаем

const char \*ip\_str = "0.0.0.0"; // слушаем все интерфейсы по умолчанию

int port = 12345;

if (argc == 2)

{

port = std::stoi(argv[1]);

}

else if (argc >= 3)

{

ip\_str = argv[1];

port = std::stoi(argv[2]);

}

std::cout << " Ожидаем пакеты на IP: " << ip\_str << ", порт: " << port << "\n";

int tap\_fd = open\_tap("tap1");

std::cout << "📡 tap1 открыт для записи расшифрованных Ethernet-кадров\n";

// Создаём UDP-сокет

int sock = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sock < 0)

{

perror("socket");

return 1;

}

sockaddr\_in local\_addr{};

local\_addr.sin\_family = AF\_INET;

local\_addr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip\_str, &local\_addr.sin\_addr) <= 0)

{

std::cerr << " Неверный IP-адрес\n";

return 1;

}

if (bind(sock, (sockaddr \*)&local\_addr, sizeof(local\_addr)) < 0)

{

perror(" bind() не удался");

return 1;

}

// std::cout << " bind() выполнен успешно\n";

// === [Автоматический обмен ключами через UDP] ===

unsigned char my\_public\_key[crypto\_kx\_PUBLICKEYBYTES];

unsigned char my\_private\_key[crypto\_kx\_SECRETKEYBYTES];

crypto\_kx\_keypair(my\_public\_key, my\_private\_key);

// 1. Принимаем публичный ключ отправителя

unsigned char sender\_public\_key[crypto\_kx\_PUBLICKEYBYTES];

sockaddr\_in sender\_addr{};

socklen\_t sender\_len = sizeof(sender\_addr);

ssize\_t received = recvfrom(sock, sender\_public\_key, crypto\_kx\_PUBLICKEYBYTES, 0,

(sockaddr \*)&sender\_addr, &sender\_len);

if (received != crypto\_kx\_PUBLICKEYBYTES)

{

std::cerr << " Ошибка при получении публичного ключа отправителя\n";

return 1;

}

std::cout << " Публичный ключ отправителя получен\n";

// 2. Отправляем свой публичный ключ обратно

sendto(sock, my\_public\_key, crypto\_kx\_PUBLICKEYBYTES, 0,

(sockaddr \*)&sender\_addr, sender\_len);

std::cout << " Отправлен свой публичный ключ отправителю\n";

// 3. Вычисляем общий ключ (rx\_key)

std::vector<unsigned char> key(KEY\_SIZE);

if (crypto\_kx\_server\_session\_keys(

key.data(), nullptr,

my\_public\_key, my\_private\_key,

sender\_public\_key) != 0)

{

std::cerr << " Ошибка при расчёте общего ключа (server)\n";

return 1;

}

// Создаём второй сокет для отправки

int send\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

if (send\_sock < 0)

{

perror("send socket");

return 1;

}

// Запускаем отправку кадров в отдельном потоке

std::thread send\_thread(send\_frames, tap\_fd, send\_sock, sender\_addr, std::ref(key));

// Основной цикл приёма

while (true)

{

unsigned char buffer[MAX\_PACKET\_SIZE];

sockaddr\_in sender\_addr{};

socklen\_t sender\_len = sizeof(sender\_addr);

// Принимаем UDP-пакет

ssize\_t nrecv = recvfrom(sock, buffer, sizeof(buffer), 0, (sockaddr \*)&sender\_addr, &sender\_len);

if (nrecv <= NONCE\_SIZE)

continue;

// Разделяем nonce и ciphertext

std::vector<unsigned char> nonce(buffer, buffer + NONCE\_SIZE);

std::vector<unsigned char> ciphertext(buffer + NONCE\_SIZE, buffer + nrecv);

// Расшифровываем

std::vector<unsigned char> decrypted(ciphertext.size());

unsigned long long decrypted\_len = 0;

int result = crypto\_aead\_chacha20poly1305\_ietf\_decrypt(

decrypted.data(), &decrypted\_len,

nullptr,

ciphertext.data(), ciphertext.size(),

nullptr, 0,

nonce.data(), key.data());

if (result != 0)

{

std::cerr << " Ошибка расшифровки!\n";

continue;

}

// Проверяем, что длина хотя бы 32 байта (под хеш)

if (decrypted\_len < HASH\_SIZE)

{

std::cerr << " Слишком маленький расшифрованный буфер!\n";

continue;

}

// Первые 32 байта — это хеш

unsigned char received\_hash[HASH\_SIZE];

std::memcpy(received\_hash, decrypted.data(), HASH\_SIZE);

// Остальная часть – это сообщение

size\_t msg\_len = decrypted\_len - HASH\_SIZE;

// Считаем свой хеш

unsigned char actual\_hash[HASH\_SIZE];

crypto\_hash\_sha256(actual\_hash,

decrypted.data() + HASH\_SIZE, // данные начинаются через 32 байта

msg\_len);

// Сравниваем

if (std::memcmp(received\_hash, actual\_hash, HASH\_SIZE) != 0)

{

std::cerr << " Хеш не совпадает — данные повреждены!\n";

continue;

}

if (message\_mode)

{

// Теперь выводим сообщение:

std::string received\_msg(

reinterpret\_cast<char \*>(decrypted.data() + HASH\_SIZE),

msg\_len);

std::cout << " Получено сообщение: "

<< msg\_len << " байт): "

<< received\_msg << "\n";

}

else

{

size\_t data\_len = decrypted\_len - HASH\_SIZE;

std::vector<unsigned char> data\_buf(data\_len);

std::memcpy(data\_buf.data(), decrypted.data() + HASH\_SIZE, data\_len);

// Пишем расшифрованный (и проверенный) кадр в tap1

write(tap\_fd, data\_buf.data(), data\_len);

std::cout << " Принят и расшифрован кадр (" << data\_len << " байт)\n";

std::cout << " Хеши совпадают — кадр корректен\n";

}

}

close(tap\_fd);

close(sock);

return 0;

}

# ПРИЛОЖЕНИЕ В - Исходный код файла test\_speed.cpp

#include <sodium.h>

#include <iostream>

#include <chrono>

#include <vector>

int main() {

if (sodium\_init() < 0) {

std::cerr << " Не удалось инициализировать libsodium\n";

return 1;

}

const size\_t MESSAGE\_SIZE = 10 \* 1024 \* 1024; // 10 МБ

std::vector<unsigned char> message(MESSAGE\_SIZE);

randombytes\_buf(message.data(), MESSAGE\_SIZE);

std::vector<unsigned char> key(crypto\_aead\_chacha20poly1305\_IETF\_KEYBYTES);

std::vector<unsigned char> nonce(crypto\_aead\_chacha20poly1305\_IETF\_NPUBBYTES);

randombytes\_buf(key.data(), key.size());

randombytes\_buf(nonce.data(), nonce.size());

std::vector<unsigned char> ciphertext(MESSAGE\_SIZE + crypto\_aead\_chacha20poly1305\_IETF\_ABYTES);

std::vector<unsigned char> decrypted(MESSAGE\_SIZE);

unsigned long long ciphertext\_len;

unsigned long long decrypted\_len;

auto start = std::chrono::high\_resolution\_clock::now();

// Шифрование

crypto\_aead\_chacha20poly1305\_ietf\_encrypt(

ciphertext.data(), &ciphertext\_len,

message.data(), message.size(),

nullptr, 0, nullptr, nonce.data(), key.data()

);

// Расшифровка с проверкой результата

int result = crypto\_aead\_chacha20poly1305\_ietf\_decrypt(

decrypted.data(), &decrypted\_len,

nullptr,

ciphertext.data(), ciphertext\_len,

nullptr, 0, nonce.data(), key.data()

);

auto end = std::chrono::high\_resolution\_clock::now();

if (result != 0) {

std::cerr << " Расшифровка не удалась (возможно, неверный ключ или повреждённые данные)\n";

return 1;

}

// Проверка совпадения данных

if (decrypted\_len != message.size() || !std::equal(message.begin(), message.end(), decrypted.begin())) {

std::cerr << " Расшифрованные данные не совпадают с исходными\n";

return 1;

}

double duration\_sec = std::chrono::duration<double>(end - start).count();

double speed\_mb\_per\_sec = MESSAGE\_SIZE / (1024.0 \* 1024.0) / duration\_sec;

std::cout << " Шифрование + расшифровка успешны\n";

std::cout << " Объём: " << MESSAGE\_SIZE / (1024 \* 1024) << " МБ\n";

std::cout << " Скорость: " << speed\_mb\_per\_sec << " МБ/с\n";

return 0;

}