Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии**»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Реализация протокола транспортного уровня с поддержкой шифрования данных

Студент группы **ИУ7-32М** _____ **А. В. Романов**
(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ **А. М. Никульшин**
(Подпись, дата) (И.О. Фамилия)

***2023** г.*

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В этом контексте шифрование данных становится критически важным для обеспечения конфиденциальности и целостности информации. Применение криптографических протоколов и алгоритмов на уровне передачи данных позволяет защитить информацию от несанкционированного доступа и обеспечить ее конфиденциальность. Шифрование данных не только предотвращает возможность прочтения или модификации данных злоумышленниками, но и гарантирует аутентификацию и целостность передаваемой информации. Целью данной курсовой работы является разработка протокола транспортного уровня с поддержкой шифрования данных.

В ходе выполнения курсового проекта необходимо решить следующие задачи:

- провести анализ предметной области;
- спроектировать протокол транспортного уровня с поддержкой шифрования;
- разработать и реализовать данный протокол.

1 Аналитическая часть

В данном разделе приводится краткий обзор предметной области. Описаны протоколы поддерживающие шифрование данных.

1.1 Обзор предметной области

1.1.1 Модель OSI

Модель OSI (Open Systems Interconnection) является концептуальным рамочным протоколом, разработанным Международной организацией по стандартизации (ISO), чтобы стандартизировать связь между различными компьютерными системами [1]. Она была определена в 1984 году и является основным принципом организации и реализации сетевых протоколов.

Модель OSI состоит из семи уровней, каждый из которых выполняет определенные функции для обеспечения надежной и эффективной коммуникации (см. рис 1).

- физический уровень: обеспечивает физическое соединение между устройствами и передачу битов по сети;
- канальный уровень: управляет надежной доставкой данных внутри локальной сети;
- сетевой уровень: обеспечивает маршрутизацию и передачу данных между различными сетями;
- транспортный уровень: отвечает за установление, управление и контроль надежной передачи данных между приложениями;
- сеансовый уровень: управляет установлением, поддержкой и завершением сеансов связи между устройствами;
- представительный уровень: обеспечивает преобразование данных в формат, понятный для приложений;
- прикладной уровень: предоставляет интерфейс для взаимодействия с приложениями.

Модель OSI широко используется при разработке и реализации сетевых

Рисунок 1 – Модель OSI

протоколов, таких как TCP/IP, Ethernet и многих других. Она обеспечивает стандартизацию и согласованность в связи между различными системами и является основополагающей моделью для понимания работы сетевых сред.

1.1.2 Транспортный уровень

Транспортный уровень является третьим уровнем в сетевой архитектуре OSI. Он отвечает за передачу данных между конечными устройствами или хостами в сети. Основной задачей транспортного уровня является обеспечение эффективной и надежной передачи данных.

На транспортном уровне происходит сегментация данных на пакеты, каждый из которых содержит адрес отправителя и получателя, а также другую

необходимую информацию. Пакеты передаются через различные узлы сети до достижения адресата.

1.1.3 Протоколы транспортного уровня

На транспортном уровне используются различные протоколы для обеспечения надежной передачи данных. Наиболее распространенными из них являются протоколы TCP (Transmission Control Protocol) [2] и UDP (User Datagram Protocol) [3].

TCP является протоколом ориентированным на соединения. Он гарантирует доставку данных в правильном порядке и с контролем ошибок.

— TCP является соединительным протоколом. Он обеспечивает надежную, ориентированную на поток передачу данных между узлами в сети.
— Для установления соединения TCP использует трехстороннее рукопожатие (three-way handshake), включающее отправку и получение пакетов SYN (synchronize) и ACK (acknowledge).
— TCP контролирует порядок пакетов и гарантирует доставку данных без потерь, дублирования или повреждений.
— Обеспечивает контроль нагрузки и управление потоком данных, чтобы избежать перегрузки сети.
— TCP имеет встроенный механизм повторной передачи и контроля ошибок, что гарантирует целостность получаемых данных.

Протокол UDP используется в приложениях, где небольшие задержки более предпочтительны, например, в потоковой передаче видео и аудио. Основные особенности данного протокола:

— UDP является безсоединительным протоколом.
— Не обеспечивает надежную доставку данных, контроль порядка пакетов или ретрансляцию потерянных пакетов.
— UDP обеспечивает минимальное накладные расходы и более быструю передачу данных за счет отсутствия механизмов, используемых в TCP.
— Он является хорошим выбором для приложений, где небольшие задержки

важны, например, в реальном времени видео или голосовой связи.

— Протокол UDP также удобен для широковещательной и многоадресной передачи данных.

— В UDP-пакете нет гарантии доставки, но он прост и эффективен в простых сценариях, где периодическое обновление информации является приемлемым, а небольшие потери данных не критичны.

1.1.4 Шифрование данных

Шифрование данных является важным аспектом безопасности при передаче информации. Оно используется для защиты данных от несанкционированного доступа и предотвращения их изменения или подделки.

Шифрование данных может быть симметричным или асимметричным. В симметричном шифровании используется один и тот же ключ для шифрования и расшифрования данных.

Примерами симметричных алгоритмов шифрования является:

- AES – Advanced Encryption Standard [4];
- DES – Data Encryption Standard [5].

В асимметричном шифровании используется пара ключей: публичный и приватный. Публичный ключ используется для шифрования данных, а приватный ключ – для их расшифровки. Это обеспечивает большую безопасность, так как приватный ключ хранится в секрете. Ниже представлены примеры наиболее популярных алгоритмов асимметричного шифрования:

- RSA – Rivest-Shamir-Adleman [6];
- ECC – Elliptic Curve Cryptography [6];
- Diffie-Hellman [6].

1.2 Протоколы транспортного уровня с поддержкой шифрования

1.2.1 Secure Sockets Layer (SSL)

SSL - это криптографический протокол для защиты передачи данных в сети. Он широко применяется в веб-браузерах, электронной почте, мгновенных сообщениях и IP-телефонии.

Протокол SSL обеспечивает следующие функции [7]:

— Безопасность: данные защищаются симметричным шифрованием от несанкционированного доступа.
— Аутентификация: можно проверить личность участников соединения с использованием асимметричного шифрования.
— Целостность: каждое сообщение содержит код аутентификации сообщения, который позволяет проверить, что данные не были изменены или потеряны в процессе передачи.

SSL работает в две фазы: рукопожатие и передача данных. Во время рукопожатия клиент и сервер используют открытый ключ для установки секретного ключа, который будет использоваться для шифрования данных во время передачи.

Рукопожатие начинается с того, что клиент отправляет «hello» сообщение серверу, содержащее список поддерживаемых клиентом алгоритмов шифрования. Сервер отвечает аналогичным «hello» сообщением, выбирая наиболее подходящий алгоритм из списка. Затем сервер отправляет свой сертификат, который содержит публичный ключ сервера.

Сертификат - это набор данных, который подтверждает подлинность. Доверенный центр сертификации генерирует и проверяет сертификат, чтобы удостовериться в его подлинности. Чтобы получить сертификат, сервер отправляет свой публичный ключ в центр сертификации по безопасному каналу. Сертификат содержит идентификаторы сервера, публичный ключ и другую информацию. Центр сертификации создает отпечаток сертификата, который является контрольной суммой, и подписывает сертификат с использованием своего приватного ключа.

Для проверки сертификата сервера клиент использует публичный ключ центра сертификации для расшифровки подписи. Затем клиент самостоятельно вычисляет отпечаток сертификата сервера и сравнивает его с расшифрованным значением. Если они не совпадают, то сертификат подделан. У клиента должен

быть доступ к публичным ключам доверенных центров сертификации. Многие браузеры имеют такие списки в своем коде. Когда сервер аутентифицирован, он использует открытый ключ для установки секретного ключа для шифрования данных.

Фаза рукопожатия заканчивается отправкой «finished» сообщений, когда обе стороны готовы использовать секретный ключ. Начинается фаза передачи данных, в которой каждая сторона разбивает сообщения на фрагменты и добавляет к ним коды аутентификации сообщений (MAC). MAC является зашифрованным отпечатком, основанным на содержимом сообщений. Он вычисляется вместе с секретным ключом во время рукопожатия. Каждая сторона объединяет данные фрагментов, коды аутентификации сообщений, заголовки и шифрует их с использованием секретного ключа, чтобы создать полный пакет SSL. При получении пакета каждая сторона расшифровывает его и сравнивает полученный код аутентификации сообщения с собственным. Если они не совпадают, то пакет был подделан. На рисунке 2 представлена концептуальная схема работы SSL.

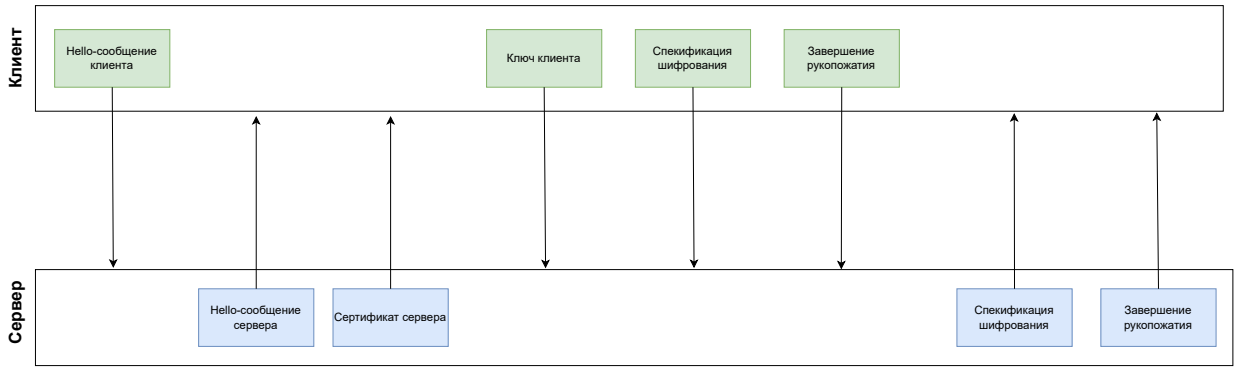

Рисунок 2 – Концептуальная схема работы SSL

SSL поддерживает 3 типа аутентификации:

1) аутентификация обеих сторон (клиент — сервер);
2) аутентификация сервера с неаутентифицированным клиентом;
3) полная анонимность.

Обычно для аутентификации используются алгоритмы: RSA и DSA.

1.2.2 Internet Protocol Security (IPSec)

IPsec (Internet Protocol Security) - набор протоколов, которые взаимодействуют для обеспечения безопасной передачи IP-пакетов по незащищенным сетям.

IP-пакет - это блок данных, который передается по компьютерной сети и состоит из заголовка (с информацией об адресе отправителя и получателя и типе данных), полезной нагрузки (информация, передаваемая пользователем) и трейлера (дополнительная информация).

Протоколы IPsec обеспечивают [8]:

— Целостность данных – защита от потери, изменения и дублирования при передаче.
— Аутентичность отправителя – гарантия, что данные были переданы от надежного источника;
— Конфиденциальность — защита зашифрованных данных от несанкционированного просмотра.

Архитектурно IPsec состоит из 4 уровней [8], представленных на рисунке 3.

На **первом уровне** находятся протоколы AH и ESP, которые гарантируют защиту данных на всех этапах их передачи. Они выполняют основные функции IPsec: аутентификацию, шифрование и являются ключевыми компонентами этой системы.

Протокол AH осуществляет:

1) проверку подлинности: пользователь может удостовериться, что взаимодействует с ожидаемыми абонентами;
2) целостность передаваемых данных: пользователь может обнаружить изменения данных в процессе передачи;
3) защиту от повторного использования данных: данные не могут быть воспроизведены злоумышленниками.

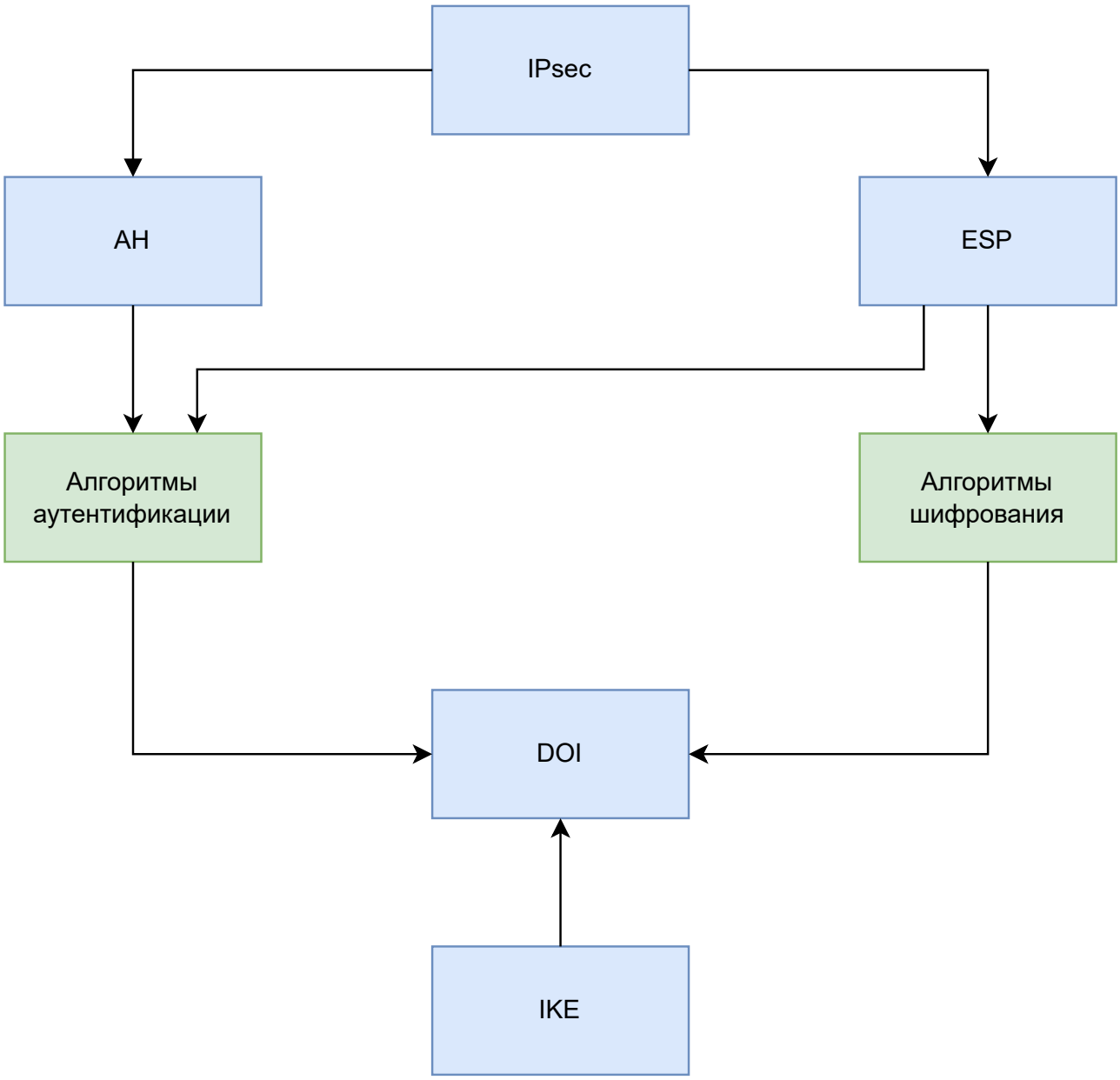

Рисунок 3 – Архитектура протокола IPSec

Протокол AH обеспечивает высокую степень защиты содержимого IP-пакета, поскольку предотвращает любые изменения в нем [9]. Однако это ограничивает его совместимость с сетевым режимом NAT (Network Address Translation), который изменяет IP-адреса при передаче IP-пакетов. Кроме того, в протоколе AH отсутствуют механизмы для защиты конфиденциальности передаваемых данных: злоумышленники не могут изменить данные, но могут их прочесть. Протокол ESP (Encapsulating Security Payload), подобно AH, поддерживает аутентификацию и целостность IP-пакета. Однако он также обеспечивает

конфиденциальность передаваемых данных через шифрование. При этом применение всех этих функций для ESP не является обязательным, однако методы шифрования всегда необходимо использовать. Обычно протоколы ESP и AH используются независимо друг от друга, однако их комбинированное применение также возможно [**?**].

Функции IPsec осуществляются конкретными алгоритмами, находящимися на **втором уровне** архитектуры: Аутентификация в IPsec реализуется при помощи специальных алгоритмов в протоколах AH и ESP. Например, для IPsec стандартные алгоритмы аутентификации HMAC используют общий секретный ключ для участников соединения [10]. Алгоритмы HMAC обязательны для протокола AH и опциональны для ESP. Подробнее о процессе аутентификации можно прочитать в разделе Аутентификация в IPsec. Протокол ESP также поддерживает различные алгоритмы шифрования, включая DES, 3DES и AES. Это дополнительно усиливает защиту IP-пакета: для получения доступа к информации необходимо не только расшифровать данные, но и определить примененные алгоритмы шифрования. Подробнее о шифровании можно прочитать в разделе Шифрование в IPsec.

На **третьем уровне** находится DOI (домен интерпретации), который содержит информацию о примененных алгоритмах и протоколах. Использование DOI обусловлено тем, что протоколы AH и ESP могут поддерживать различные алгоритмы.

На **четвертом уровне** находится протокол IKE (Internet Key Exchange), который является основой IPsec. Он используется для установления политики безопасности соединения, таким образом отправитель и получатель согласовывают алгоритмы аутентификации и шифрования [11]. Кроме того, этот протокол обеспечивает генерацию и распределение ключей между участниками защищенного соединения.

2 Конструкторская часть

В этом разделе представлена архитектура разработанного протокола. Описаны используемые структуры данных.

2.1 Архитектура протокола

2.1.1 Описание протокола

На рисунке 4 представлена концептуальная схема работы протокола.

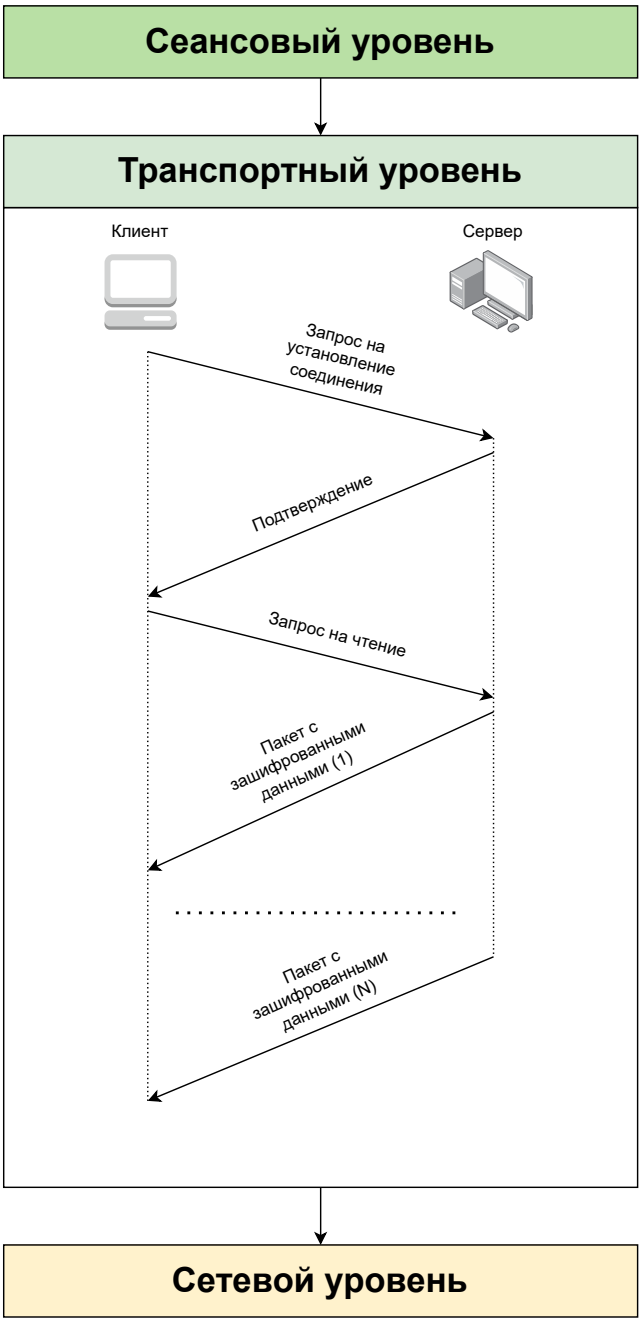

Рисунок 4 – Концептуальная схема работы протокола

Шаги работы протокола можно описать следующим образом:

1) клиент отправляет серверу запрос на установления соединения;
2) сервер отправляет ответ-подтверждение;
3) если подтверждение на установление соединения не получено, клиент завершает свою работу;
4) в случае, если подтверждение получено, клиент отправляет запрос на чтение данных;
5) сервер подготавливает пакеты к отправке, предварительно зашифровывая передаваемые данные;
6) сервер отправляет клиенту пакеты с зашифрованными данными;
7) клиент расшифровывает полученные от сервера данные;

Размер передаваемых данных должен обязательно кратен блоку шифрования. В обратном случае, сервер дополняет их необходимым количеством байт, состоящим из нулей.

2.1.2 Шифрование данных

Для шифрования данных используется симметричный алгоритм AES-192. Это означает, что один и тот же ключ используется и для шифрования, и для расшифровки отправляемых данных. Протокол предполагает, что ключ не передаётся по сети. Сервер и клиент должны самостоятельно определить, какой ключ шифрования необходимо использовать для шифрования и расшифровывания данных.

2.2 Структура пакета

2.2.1 Описание пакета

Пакет, используемый в проектируемом протоколе, можно разделить на три логических части:

1) заголовок IP – метаданные, необходимые для сетевого уровня OSI.
2) метаданные пакета – информация о передаваемых данных;
3) полезная нагрузка – непосредственно сами данные.

Описание структуры пакета представлено на рисунке 5.

Рисунок 5 – Структура пакета

Под заголовок IP выделяется первые 20 байт, а для метаданных следующие восемь байт. Остальная память используется для хранения данных, размер которые не может превышать 65488 байт и должен быть кратен размеру блоку шифрования (16 байт). Зашифрованными передаются только сами данные, содержимое IP заголовка и метаданных передается в не зашифрованном виде.

Для каждого отправляемого пакета вычисляется контрольная сумма, ко-

торая является суммой хэш-сумм IP-заголовка, метаданных и данных (в зашифрованном виде).

2.2.2 Метаданные

На рисунке 6 представлено описание метаданных пакета.

Рисунок 6 – Метаданные пакета

— В первых 16-ти битах хранится адрес отправителя пакета.
— Биты с 16 по 31 содержат адрес получателя.
— Биты с 32 по 47 – размер зашифрованных данных.
— Биты с 48 по 63 – контрольная сумма пакета.

3 Технологическая часть

В данном разделе описываются средства разработки программного обеспечения и его сборка.

3.1 Выбор средств разработки

3.1.1 Ядро Linux

Разрабатываемый протокол был реализован на уровне ядра ОС Linux. Данный выбор был обусловлен несколькими факторами:

— Производительность: реализация шифрования на уровне ядра позволяет использовать аппаратные возможности процессора и обеспечивать высокую скорость обработки криптографических операций.

— Универсальность: реализация на уровне ядра ОС позволяет использовать его в различных сетевых приложениях, независимо от их конкретной реализации.

— Взаимодействие с другими слоями сетевой модели: более тесное взаимодействие протокола транспортного уровня с другими слоями сетевой модели OSI, такими как сетевой уровень (IP) и канальный уровень (Ethernet), что может улучшить общую эффективность и производительность сети.

— Открытый исходный код ядра Linux.

3.1.2 Выбор языка программирования

Для реализации протокола был выбран язык программирования C с использованием стандарта C11 [12]. Данный выбор обоснован тем, что язык C является основным языком, на котором написано ядро Linux.

3.2 Интерфейс сокетов

Протокол был реализован с помощью интерфейса сокетов. В ядре Linux создание сокетов реализовано с использование системного вызова `socket`, который требует трех обязательных аргументов:

1) семейство адресов сокета;
2) тип сокетов;

3) используемый протокол.

В качестве семейства адресов используется `AF_INET`, тип сокетов – `SOCK_DGRAM`. Таким образом, для того чтобы создать необходимый сокет с разработанным протоколом, нужно выполнить системный вызов `socket`. Пример вызова представлен в листинге 1.

Листинг 1: Пример создания сокета

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_ENCRYPTED);
```

3.3 Сборка программного обеспечения

3.3.1 Прошивка устройства

Сборка прошивки производится с помощью кросс-платформенной системы сборки buildroot [13]. В листинге 2 представлена конфигурация, которая должна быть использована в процессе сборки прошивки устройства.

Листинг 2: Конфигурационный файл для сборки прошивки устройства

```
BR2_aarch64=y
BR2_cortex_a72=y
BR2_ARM_FPU_VFPV4=y
BR2_KERNEL_HEADERS_6_6=y
BR2_TOOLCHAIN_BUILDROOT_CXX=y
BR2_PACKAGE_OVERRIDE_FILE="board/rpi4-64/override_module.mk"
BR2_SYSTEM_DHCP="eth0"
BR2_ROOTFS_POST_BUILD_SCRIPT="board/rpi4-64/post-build.sh"
BR2_ROOTFS_POST_IMAGE_SCRIPT="board/rpi4-64/post-image.sh"
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_GIT=y
BR2_LINUX_KERNEL_CUSTOM_REPO_URL="file://$(TOPDIR)/../linux"
BR2_LINUX_KERNEL_CUSTOM_REPO_VERSION="my-rpi-6.6.y"
BR2_LINUX_KERNEL_DEFCONFIG="bcm2711"
```

```
BR2_LINUX_KERNEL_DTS_SUPPORT=y
BR2_LINUX_KERNEL_INTREE_DTS_NAME="broadcom/bcm2711-rpi-4-b"
BR2_LINUX_KERNEL_NEEDS_HOST_OPENSSL=y
BR2_PACKAGE_RPI_FIRMWARE=y
BR2_PACKAGE_RPI_FIRMWARE_VARIANT_PI4=y
BR2_PACKAGE_RPI_FIRMWARE_CONFIG_FILE="board/rpi4-64/64bit.txt"
BR2_TARGET_ROOTFS_EXT2=y
BR2_TARGET_ROOTFS_EXT2_4=y
BR2_TARGET_ROOTFS_EXT2_SIZE="120M"
# BR2_TARGET_ROOTFS_TAR is not set
BR2_PACKAGE_HOST_DOSFSTOOLS=y
BR2_PACKAGE_HOST_GENIMAGE=y
BR2_PACKAGE_HOST_MTOOLS=y
BR2_PACKAGE_ENCRYPTED_PROTO_EXAMPLE=y
```

Прошивка подойдет для устройств Raspberry Pi 4 Model B.

3.3.2 Сборка ядра Linux

Ядро Linux собирается с помощью buildroot в виде пакета `linux`. Используемая версия ядра – 6.6.

Сам протокол реализован в виде исходного файлов ядра, находящегося по адресу `net/ipv4/encrypted.c`. Это означает о том, что он собирается в составе Linux, а не как отдельный загружаемый модуль ядра. Для этого, был модифицирован один из Makefile ядра. Модификация представлена в листинге 3.

Листинг 3: Изменения в `net/ipv4/Makefile`

```
obj-y += encrypted.o
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была достигнута его цель – разработан протокола транспортного уровня с поддержкой шифрования данных. Были решены следующие задачи:

- проведен анализ предметной области;
- спроектирован протокол транспортного уровня с поддержкой шифрования;
- разработан и реализован данный протокол.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Networks | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/no/aix/7.1?topic=networks – (01.11.2023)

2. Протоколы TCP/IP | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/ru/aix/7.2?topic=protocol-tcpip-protocols – (02.11.2023)

3. User Datagram Protocol (UDP) | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/zvm/7.2?topic=protocols-user-datagram-protocol-udp – (11.11.2023)

4. Advanced Encryption Standard (AES) | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/zos/2.1.0?topic=data-advanced-encryption-standard-aes – (17.11.2023)

5. What is DES and AES? | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/zos/2.1.0?topic=encryption-what-is-des-aes – (17.11.2023)

6. Cryptography concepts | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/i/7.1?topic=cryptography-concepts – (17.11.2023)

7. Secure Sockets Layer (SSL) protocol | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/ibm-http-server/9.0.5?topic=communications-secure-sockets-layer-ssl-protocol – (17.11.2023)

8. Overview of IPSec | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/pt/zos/2.4.0?topic=folder-overview-ipsec – (17.11.2023)

9. AH and ESP protocols | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/zos/2.2.0?topic=ipsec-ah-esp-protocols – (01.12.2023)

10. HMAC Generate | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/en/zos/2.4.0?topic=messages-hmac-generate-csnbhmg-csnbhmg1-csnehmg-csnehmg1 – (03.12.2023)

11. Возможности протокола Internet Key Exchange (IKE) | IBM. [Электронный ресурс] – Режим доступа: https://www.ibm.com/docs/ru/aix/7.2?topic=features-internet-key-exchange – (06.12.2023)

12. ISO/IEC 9899:201x [Электронный ресурс] – Режим доступа: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf – (06.12.2023)

13. Buildroot - Making Embedded Linux Easy [Электронный ресурс] – Режим доступа: https://buildroot.org/ – (06.12.2023)

ПРИЛОЖЕНИЕ А

Листинг 4: Исходный код разработанного протокола

```c
// SPDX-License-Identifier: GPL-2.0-or-later

#include <net/sock.h>
#include <net/protocol.h>
#include <net/inet_common.h>
#include <net/inet_hashtables.h>
#include <net/encrypted.h>
#include <net/ip.h>
#include <crypto/skcipher.h>
#include <linux/socket.h>
#include <linux/net.h>
#include <linux/skbuff.h>
#include <linux/crypto.h>
#include <linux/scatterlist.h>

#define CIPHER_ALGO "aes"

static inline void encrypted_close(struct sock *sk, long timeout)
{
  sk_common_release(sk);
}

static int encrypted_disconnect(struct sock *sk, int flags)
{
  struct inet_sock *inet = inet_sk(sk);

  sk->sk_state = TCP_CLOSE;
  inet->inet_daddr = 0;
  inet->inet_dport = 0;

  sk_dst_reset(sk);
```

```
32
33    return 0;
34  }
35
36  static int encrypted_sk_init(struct sock *sk)
37  {
38    (void)sk;
39
40    return 0;
41  }
42
43  static void encrypted_sk_destroy(struct sock *sk)
44  {
45    release_sock(sk);
46  }
47
48  static int encrypt_msghdr_data(struct msghdr *msg, size_t size)
49  {
50    struct crypto_skcipher *skcipher = NULL;
51    struct skcipher_request *req = NULL;
52    char key[] = "0123456789ABCDEF";
53    char iv[] = "1234567890ABCDEF";
54    char *buffer, *out;
55    int ret = 0;
56
57    buffer = kmalloc(size, GFP_KERNEL);
58    if (unlikely(!buffer))
59      return -ENOMEM;
60
61    out = kmalloc(size, GFP_KERNEL);
62    if (unlikely(!buffer)) {
63      ret = -ENOMEM;
64      goto out;
65    }
66
```

```
67    skcipher = crypto_alloc_skcipher(CIPHER_ALGO, 0, 0);
68    if (IS_ERR(skcipher)) {
69      ret = -ENOMEM;
70      goto out;
71    }
72
73    ret = crypto_skcipher_setkey(skcipher, key, sizeof(key));
74    if (ret)
75      goto out;
76
77    req = skcipher_request_alloc(skcipher, GFP_KERNEL);
78    if (!req) {
79      ret = -ENOMEM;
80      goto out;
81    }
82
83    memcpy_from_msg(buffer, msg, size);
84
85    sg_init_one(req->dst, out, sizeof(out));
86    sg_init_one(req->src, buffer, sizeof(buffer));
87    skcipher_request_set_crypt(req, req->src, req->dst, sizeof(buffer), i
88
89    ret = crypto_skcipher_encrypt(req);
90    if (ret)
91      goto out;
92
93    memcpy_to_msg(msg, out, req->cryptlen);
94
95  out:
96    if (req)
97      skcipher_request_free(req);
98
99    if (skcipher)
100     crypto_free_skcipher(skcipher);
101
```

```
102   kfree(out);
103   kfree(buffer);
104
105   return ret;
106 }
107
108 static int decrypt_msghdr_data(struct msghdr *msg, size_t size)
109 {
110   struct crypto_skcipher *skcipher = NULL;
111   struct skcipher_request *req = NULL;
112   char key[] = "0123456789ABCDEF";
113   char iv[] = "1234567890ABCDEF";
114   char *buffer, *out;
115   int ret = 0;
116
117   buffer = kmalloc(size, GFP_KERNEL);
118   if (unlikely(!buffer))
119     return -ENOMEM;
120
121   out = kmalloc(size, GFP_KERNEL);
122   if (unlikely(!buffer)) {
123     ret = -ENOMEM;
124     goto out;
125   }
126
127   memcpy_from_msg(buffer, msg, size);
128
129   skcipher = crypto_alloc_skcipher(CIPHER_ALGO, 0, 0);
130   if (IS_ERR(skcipher)) {
131     ret = -ENOMEM;
132     goto out;
133   }
134
135   ret = crypto_skcipher_setkey(skcipher, key, sizeof(key));
136   if (ret)
```

```
137     goto out;
138
139   req = skcipher_request_alloc(skcipher, GFP_KERNEL);
140   if (!req) {
141     ret = -ENOMEM;
142     goto out;
143   }
144
145   memcpy_from_msg(buffer, msg, size);
146
147   sg_init_one(req->dst, out, sizeof(out));
148   sg_init_one(req->src, buffer, sizeof(buffer));
149   skcipher_request_set_crypt(req, req->src, req->dst, sizeof(buffer), i
150
151   ret = crypto_skcipher_encrypt(req);
152   if (ret)
153     goto out;
154
155   memcpy_to_msg(msg, out, size);
156
157 out:
158   if (req)
159     skcipher_request_free(req);
160
161   if (skcipher)
162     crypto_free_skcipher(skcipher);
163
164   kfree(out);
165   kfree(buffer);
166
167   return ret;
168 }
169
170 static int encrypted_sendmsg(struct sock *sk, struct msghdr *msg, size_
171 {
```

```
172   struct inet_sock *inet = inet_sk(sk);
173   struct net *net = sock_net(sk);
174   DECLARE_SOCKADDR(struct sockaddr_in *, usin, msg->msg_name);
175   struct ipcm_cookie ipc;
176   struct flowi4 *fl4;
177   struct flowi4 fl4_stack;
178   struct rtable *rt = NULL;
179   __be32 daddr, faddr, saddr;
180   __u8 flow_flags;
181   __be16 dport;
182   u8 tos, scope;
183   size_t ulen = len;
184   int err;
185
186   ulen += sizeof(encrypted_hdr);
187
188   if (usin) {
189     daddr = usin->sin_addr.s_addr;
190     dport = usin->sin_port;
191
192     if (!dport)
193       return -EINVAL;
194   }
195
196   ipcm_init_sk(&ipc, inet);
197
198   saddr = ipc.addr;
199   ipc.addr = faddr = daddr;
200
201   if (ipc.opt && ipc.opt->opt.srr)
202     faddr = ipc.opt->opt.faddr;
203
204   tos = get_rttos(&ipc, inet);
205   scope = ip_sendmsg_scope(inet, &ipc, msg);
206
```

```
207   if (ipv4_is_multicast(daddr)) {
208     if (!ipc.oif || netif_index_is_l3_master(sock_net(sk), ipc.oif))
209       ipc.oif = inet->mc_index;
210     if (!saddr)
211       saddr = inet->mc_addr;
212   } else if (!ipc.oif) {
213     ipc.oif = inet->uc_index;
214   } else if (ipv4_is_lbcast(daddr) && inet->uc_index) {
215     if (ipc.oif != inet->uc_index &&
216         ipc.oif == l3mdev_master_ifindex_by_index(sock_net(sk),
217                     inet->uc_index)) {
218       ipc.oif = inet->uc_index;
219     }
220   }
221
222   flow_flags = inet_sk_flowi_flags(sk);
223   fl4 = &fl4_stack;
224   flowi4_init_output(fl4, ipc.oif, ipc.sockc.mark, tos, scope,
225       sk->sk_protocol, flow_flags, faddr, saddr,
226       dport, inet->inet_sport, sk->sk_uid);
227
228   rt = ip_route_output_flow(net, fl4, sk);
229   if (IS_ERR(rt)) {
230     pr_err("encrypted_socket: failed to ip_route_output_flow...");
231     return PTR_ERR(rt);
232   }
233
234   err = encrypt_msghdr_data(msg, len);
235   if (unlikely(err))
236     return err;
237
238   sk_dst_set(sk, dst_clone(&rt->dst));
239   ip_rt_put(rt);
240
241   return len;
```

```
242 }
243
244 static inline __sum16 encrypted_checksum(struct sk_buff *skb)
245 {
246   return __skb_checksum_complete(skb);
247 }
248
249 static int encrypted_recvmsg(struct sock *sk, struct msghdr *msg, size_
250             int flags, int *addr_len)
251 {
252   struct encrypted_hdr *hdr;
253   struct sk_buff *skb;
254   unsigned int off, copied, ulen = 0;
255   int err;
256
257   skb = skb_recv_datagram(sk, flags, &err);
258   if (!skb)
259     return 0;
260
261   hdr = encrypted_hdr(skb);
262
263   if (encrypted_checksum(skb) != hdr->checksum) {
264     pr_err("encrypted_socket: invalid checksum\n");
265     msg->msg_flags &= ~MSG_TRUNC;
266     return -EINVAL;
267   }
268
269   ulen = skb->len;
270   copied = len;
271   if (copied > ulen - off)
272     copied = ulen - off;
273
274   off = sk_peek_offset(sk, flags);
275   err = skb_copy_datagram_msg(skb, off, msg, copied);
276   if (unlikely(err))
```

```
277     return err;
278
279   err = decrypt_msghdr_data(msg, copied);
280   if (unlikely(err))
281     return err;
282
283   return copied;
284 }
285
286 struct proto encrypted_prot = {
287   .name        = "ENCRYPTED",
288   .owner        = THIS_MODULE,
289   .close        = encrypted_close,
290   .connect      = ip4_datagram_connect,
291   .disconnect      = encrypted_disconnect,
292   .init        = encrypted_sk_init,
293   .destroy      = encrypted_sk_destroy,
294   .sendmsg      = encrypted_sendmsg,
295   .recvmsg      = encrypted_recvmsg,
296   .obj_size      = sizeof(struct encrypted_sock),
297 };
298
299 static const struct proto_ops encrypted_ops = {
300   .family        = PF_INET,
301   .owner        = THIS_MODULE,
302   .release      = inet_release,
303   .bind        = inet_bind,
304   .connect      = inet_dgram_connect,
305   .socketpair      = sock_no_socketpair,
306   .accept        = sock_no_accept,
307   .poll        = datagram_poll,
308   .ioctl        = inet_ioctl,
309   .gettstamp      = sock_gettstamp,
310   .listen        = sock_no_listen,
311   .shutdown      = inet_shutdown,
```

```
312   .setsockopt      = sock_common_setsockopt,
313   .getsockopt      = sock_common_getsockopt,
314   .sendmsg      = inet_sendmsg,
315   .recvmsg      = sock_common_recvmsg,
316   .mmap         = sock_no_mmap,
317 };
318
319 static int encrypted_err(struct sk_buff *skb, u32 info)
320 {
321   return 0;
322 }
323
324 static int encrypted_rcv(struct sk_buff *skb)
325 {
326   struct encrypted_hdr *hdr = encrypted_hdr(skb);
327   struct net *net = dev_net(skb->dev);
328   unsigned short ulen;
329   __be32 saddr, daddr;
330   bool refcounted;
331   struct sock *sk;
332
333   ulen = ntohs(hdr->len);
334   saddr = ip_hdr(skb)->saddr;
335   daddr = ip_hdr(skb)->daddr;
336
337   hdr->checksum = encrypted_checksum(skb);
338
339   sk = inet_steal_sock(net, skb, sizeof(struct encrypted_hdr), saddr, h
340       daddr, hdr->dst, &refcounted, NULL);
341   if (IS_ERR(sk)) {
342     pr_err("Failed to steal socket\n");
343     return -EINVAL;
344   }
345
346   if (sk)
```

```
347     __skb_queue_tail(&sk->sk_receive_queue, skb);
348
349   return 0;
350 }
351
352 static const struct net_protocol encrypted_protocol = {
353   .handler   = encrypted_rcv,
354   .err_handler   = encrypted_err,
355   .no_policy   = 1,
356 };
357
358 static struct inet_protosw encrypted_protosw = {
359   .type     =  SOCK_DGRAM,
360   .protocol   =  IPPROTO_ENCRYPTED,
361   .prot     =  &encrypted_prot,
362   .ops     =  &encrypted_ops,
363   .flags     =  INET_PROTOSW_PERMANENT,
364 };
365
366 void __init encrypted_register(void)
367 {
368   if (proto_register(&encrypted_prot, 1)) {
369     pr_err("Failed to register encrypted protocol");
370     return;
371   }
372
373   if (inet_add_protocol(&encrypted_protocol, IPPROTO_ENCRYPTED) < 0)
374     goto out;
375
376   inet_register_protosw(&encrypted_protosw);
377
378   pr_info("Encrypted protocol successfully registered!\n");
379
380   return;
381
```

```
382 out:
383   proto_unregister(&encrypted_prot);
384 }
```