ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Московский институт электроники и математики им. А.Н. Тихонова

Чудников Александр Александрович

ИССЛЕДОВАНИЕ РЕАЛИЗАЦИИ ВИРТУАЛЬНЫХ КРИПТОГРАФИЧЕСКИХ СЕТЕВЫХ ИНТЕРФЕЙСОВ В ЯДРЕ ОС LINUX

Выпускная квалификационная работа
по специальности 10.05.01 «Компьютерная безопасность»
студента образовательной программы специалитета
«Компьютерная безопасность»

Студент

_____ А. А. Чудников
(подпись) *(И.О. Фамилия)*

Научный руководитель
к.ф.-м.н., доцент

_____ А.Ю. Нестеренко
(подпись) *(И.О. Фамилия)*

Рецензент

_____ Т. В. Малин
(подпись) *(И.О. Фамилия)*

Москва, 2026

Аннотация

Тема: Исследование реализации виртуальных криптографических сетевых интерфейсов в ядре ОС Linux.

Цель работы: Целью работы является исследование механизмов ядра семейства операционной системы `Linux` с целью последующей реализации виртуальных сетевых интерфейсов на базе криптографических протоколов, описанных в государственных стандартах (ГОСТ).

Поставленные задачи:

- исследование методов реализации виртуальных сетевых интерфейсов в ядре Linux;
- анализ криптографической подсистемы ядра «`Crypto API`» на возможность встраивания криптографических протоколов, описанных в ГОСТ;
- изучение существующих виртуальных сетевых интерфейсов;
- выбор оптимального набора криптографических протоколов;
- разработка и тестирование прототипа виртуального криптоинтерфейса.

Полученные результаты: В ходе выполнения работы было принято решение об использовании `WireGuard` в качестве основы для встраивания криптографических протоколов, описанных в ГОСТ. Был успешно разработан набор базовых криптографических примитивов в виде загружаемых модулей ядра (`LKM`) с использованием `Crypto API`. На базе архитектуры `WireGuard` разработан модуль `WireGost`, использующий отечественные криптографические алгоритмы.

Тестирование показало, что `WireGost` способен работать на скоростях вплоть до **900 Мбит/с**. Это подтверждает преимущество реализации в ядре перед решениями в пространстве пользователя, тем не менее уступает оригинальному `WireGuard`. Наиболее медленной частью системы стал блочный шифр «Кузнечик», который, по сравнению с «`ChaCha20`», используемым в оригинальном протоколе, является гораздо более дорогим с точки зрения вычислительной сложности.

Предложенные рекомендации: Для повышения производительности разработанного решения и приближения его показателей к скорости физического канала рекомендуется пересмотреть методы реализации блочного шифра «Кузнечик» в виде загружаемых модулей ядра, а также обеспечить более эффективную поддержку базовых математических операций используемых в ГОСТ.

Abstract

Topic: Research on the implementation of virtual cryptographic network interfaces in the Linux OS kernel.

Purpose: The purpose of the work is to study the mechanisms of the Linux operating system family kernel with the aim of the subsequent implementation of virtual network interfaces based on cryptographic protocols described in state standards (GOST).

Tasks:

- research of methods for implementing virtual network interfaces in the Linux kernel;
- analysis of the kernel cryptographic subsystem "`Crypto API`" regarding the possibility of embedding cryptographic protocols described in GOST;
- study of existing virtual network interfaces;
- selection of an optimal set of cryptographic protocols;
- development and testing of a virtual crypto-interface prototype.

Results: While carrying out the research the author decided to use `WireGuard` as a basis for embedding cryptographic protocols described in GOST. A set of basic cryptographic primitives was successfully developed in the form of Loadable Kernel Modules (`LKM`) using the `Crypto API`. Based on the `WireGuard` architecture, the `WireGost` module was developed using domestic cryptographic algorithms.

Testing showed that `WireGost` is capable of operating at speeds up to **900 Mbps**. This confirms the advantage of in-kernel implementation over user-space solutions; nevertheless, it yields to the original `WireGuard`. The slowest part of the system proved to be the "Kuznyechik" block cipher, which, compared to "`ChaCha20`" used in the original protocol, is much more expensive in terms of computational complexity.

Recommendations: To improve the performance of the developed solution and bring its metrics closer to the physical link speed, it is recommended to revise the implementation methods of the "Kuznyechik" block cipher in the form of loadable kernel modules, as well as to ensure more efficient support for basic mathematical operations used in GOST.

Содержание

Введение **1**
Актуальность темы исследования 1
Научная новизна 1
Объект исследования 2
Предмет исследования 2
Степень научной разработанности темы 2
Цель дипломной работы 3
Методологическая основа исследования 3
Структура работы 4

1 Теоретическая часть **5**
1.1 Основные понятия 5
1.1.1 Средство криптографической защиты информации 5
1.1.2 Виртуальная частная сеть 6
1.1.3 Сетевой адаптер 7
1.1.4 Сетевой интерфейс 7
1.1.5 Виртуальный сетевой интерфейс 8
1.1.6 Виртуальный криптографический сетевой интерфейс 8
1.1.7 Эталонная модель взаимодействия открытых систем 8
1.1.8 Модель TCP/IP 10
1.2 Теоретические основы 12
1.2.1 Сетевой стек операционной системы Linux 12
1.2.2 Краткий анализ принципов работы WireGuard 13
1.3 Модули ядра `Linux` 16
1.4 Криптографическая подсистема ядра `Linux` 16
1.4.1 Классификация криптографических примитивов 17
1.4.2 Протокол `Netlink` 19

2 Аналитическая часть **21**
2.1 Разработка и эксплуатация модулей ядра 21
2.1.1 Структура и программная реализация 21
2.1.2 Система сборки Kbuild 22
2.1.3 Управление загрузкой 23

2.1.4 Методы отладки в пространстве ядра 23
2.2 Архитектура и регистрация криптографических модулей 25
2.2.1 Структура дескриптора алгоритма 25
2.2.2 Классификация криптографических примитивов 26
2.3 Анализ криптографической архитектуры WireGuard 32
2.3.1 `Noise Framework` 32
2.4 Обзор отечественных криптографических стандартов 36
2.4.1 Алгоритм хэширования ГОСТ Р 34.11-2012 «Стрибог» 37
2.4.2 Блочный шифр ГОСТ Р 34.12-2015 «Кузнечик» 39
2.4.3 Режим MGM (Multilinear Galois Mode) 40
2.4.4 Протокол выработки общего ключа VKO ГОСТ Р 34.10-2012 41

3 Программная реализация и тестирование 43
3.1 Реализация криптографических примитивов ГОСТ 43
3.1.1 Алгоритм хэширования ГОСТ Р 34.11-2012 43
3.1.2 Конструкция HMAC на базе ГОСТ Р 34.11-2012 46
3.1.3 Блочный шифр «Кузнечик» и режим MGM 48
3.1.4 Протокол выработки общего ключа VKO ГОСТ Р 34.10-2012 52
3.2 Интеграция в сетевой протокол WireGuard 55
3.2.1 Адаптация структур данных 55
3.2.2 Модификация машины состояний Noise 55
3.2.3 Взаимодействие через Netlink 56
3.2.4 Инициализация и управление зависимостями 56
3.2.5 Конфигурация протокола Noise_GOST 58
3.3 Разработка инструментария конфигурации и тестирования 63
3.3.1 Архитектура утилиты конфигурации 63
3.3.2 Инициализация интерфейса и генерация ключей 64
3.3.3 Настройка пиров 65
3.4 Методология тестирования 66
3.4.1 Тестовая среда и топология 66
3.4.2 Архитектура тестового комплекса 67
3.5 Результаты тестирования 70
3.5.1 Эталонные показатели (Baseline) 70

3.5.2 Сравнительный анализ реализаций WireGuard 71
3.6 Выводы по главе 73

4 Заключение 74
4.1 Текущие результаты 74
4.2 Дальнейшее направление развития проекта 75

Введение

Выпускная квалификационная работа посвящена исследованию и практической реализации виртуального криптографического сетевого интерфейса в пространстве ядра операционной системы `Linux`, с использованием отечественных криптографических стандартов.

Актуальность темы исследования

Распространение практики использования криптографических стандартов, разработанных и стандартизированных на территории недружественных нашему государству стран, является одной из главных проблем отечественной криптографии. Данная проблема становится наиболее актуальной в связи с геополитическими изменениями, происходящими в мире на данный момент. Использование таких стандартов влечет за собой реальные риски для безопасности государственных и коммерческих систем, так как никто не может гарантировать, что в них не были заложены как программные закладки, так и не задокументированные возможности, как, например, в генераторе псевдослучайных чисел Dual EC. [1] Единственной надежной защитой от такого рода угроз становится повсеместный переход на отечественные криптографические стандарты.

Однако, в таком случае появляется другая проблема: практически полное отсутствие каких-либо отечественных аналогов довольно большого пласта ПО, от которого зависят как коммерческие, так и государственные системы. Данная работа в первую очередь стремится к поиску методов решения такого технического отставания в сфере ПО для поднятия виртуальных частных сетей. В работе рассматривается возможность реализации такого протокола и рассматриваются проблемы, связанные с разработкой. В качестве основы был выбран зарубежный протокол `WireGuard`, так как его реализация значительно проще и быстрее его прямых аналогов. Более того, отечественные стандарты обладают аналогами всех криптографических алгоритмов, используемых в данном протоколе.

Научная новизна

Предпринимаемые ранее попытки переноса WireGuard на отечественные криптографические стандарты [2] основывались на идее реализации userspace-версии протокола, которая является значительно более медленной. В данной работе предлагается реализация kernel-версии протокола, которая должна быть значительно быстрее и эффективнее.

Объект исследования

Объектом исследования в данной работе выступают ядро операционной системы Linux, сетевой стек ядра, криптографический API (Crypto API), а также механизмы передачи информации из пространства пользователя в ядро, такие как протокол «Netlink».

Предмет исследования

Предметом исследования выступают конкретные механизмы, архитектурные решения и программные интерфейсы, используемые в WireGuard, а также методы интеграции в нее отечественных криптографических протоколов.

Степень научной разработанности темы

Наиболее глубоко исследованной частью данной работы являются вопросы касающиеся реализации загружаемых модулей ядра Linux, а также его криптографической подсистемы (Crypto API). Данные вопросы подробно описаны как в технической документации ядра Linux, так и в многочисленных статьях и публикациях. [9; 11; 12]

Архитектура WireGuard и используемые им криптографические протоколы, подробно описаны его создателем Джейсоном Доненфилдом [3], а также отражены в работе «Analysis of the WireGuard protocol» Питера By. [20]

Сама идея переноса протокола WireGuard на криптографические алгоритмы, стандартизированные в ГОСТ (Государственный стандарт), также не яв-

ляется новой: схожее исследование с неутешительными результатами было проведено компанией «BI.ZONE» в рамках проекта «RuWireGuard» [2]. Отличительной особенностью данного проекта стала его невероятно медленная реализация, которая была выполнена на языке «Go» и функционировала в пространстве пользователя (userspace), что значительно снижало её производительность

Цель дипломной работы

Целью дипломной работы является исследование возможности встраивания в ядро Linux виртуального криптоинтерфейса, поддерживающего отечественные криптографические алгоритмы.

Методологическая основа исследования

Аналитическая часть дипломной работы представляет собой системный анализ, изучение и обобщение информации, касающейся затрагиваемых данной работой тем. Анализируемая информация бралась из технической документации, спецификаций протоколов, исследовательских работ, а также из результатов анализа исходного кода программных решений, упомянутых в аналитической части.

Практическая часть дипломной работы представляет собой разработку прототипов различных криптографических модулей ядра Linux, а также виртуального криптоинтерфейса на основе архитектуры WireGuard. Сами модули были разработаны с использованием языка программирования «C».

Исследовательская часть дипломной работы представляет собой приведенный ниже алгоритм, по которому производился сравнительный анализ интересующих нас решений:

1. **Создание среды тестирования**: создание виртуальной сети с использованием Linux Network Namespaces (netns) и виртуальных пар Ethernet (veth);
2. **Измерение базовых показателей канала**: измерение характеристик

«голого» канала без шифрования и туннелирования. С помощью утилит *iperf3* и *ping* замеряется максимальная пропускная способность (TCP/UDP), задержка (RTT) и базовое потребление ресурсов ЦП, которые принимаются за теоретический максимум;

3. **Тестирование существующих решений**: проведение измерений для оригинального протокола `WireGuard` (`ChaCha20-Poly1305`) и `userspace`-реализации протокола с отечественной криптографией (`RuWireGuard` на `Go`);

4. **Развертывание разработанного прототипа**: загрузка разработанного модуля ядра `Linux` (`WireGost`), настройка виртуальных интерфейсов и установление защищенного соединения с использованием алгоритмов ГОСТ;

5. **Комплексное нагрузочное тестирование**: измерение пропускной способности, джиттера и нагрузки на процессор при прохождении трафика через разработанный туннель.

6. **Сравнительный анализ**: сопоставление полученных результатов с показателями `userspace`-решения и оригинального WireGuard. Оценка накладных расходов на шифрование ГОСТ и расчет прироста производительности, достигнутого за счет переноса реализации в пространство ядра.

Структура работы

Дипломная работа состоит из введения, трех глав, заключения и списка использованных источников:

1. **Введение**: описание целей, задач и актуальности работы. Разбор структуры дипломной работы;

2. **Теоретические основы**: обзор базовых определений, сетевых и криптографических подсистем ядра `Linux`, а также технологий и протоколов, затронутых в работе;

3. **Архитектура ядра и WireGuard**: анализ архитектуры ядра `OS Linux`, архитектуры `WireGuard` и существующих решений на базе отечественных стандартов;

4. **Разработка и экспериментальная оценка**: описание процесса разработки прототипа и результаты экспериментальной оценки его производительности;

5. **Заключение**: в заключении подводятся итоги проделанной работы и формулируются основные выводы.

1 Теоретическая часть

1.1 Основные понятия

Основная часть дипломной работы построена на основе термина ”виртуальный криптографический сетевой интерфейс”, который в свою очередь является составным. Для того, чтобы дать ему определение, первоочередно следует определить три ключевых понятия, которые он включает:

- криптографическое средство защиты информации (СКЗИ);
- виртуальная частная сеть (VPN);
- виртуальный сетевой интерфейс (VNI).

1.1.1. Средство криптографической защиты информации

В соответствии с ГОСТ Р 50922-2006 ”Защита информации. Основные термины и определения”, под средством криптографической защиты информации (СКЗИ) понимается средство защиты информации, реализующее алгоритмы криптографического преобразования информации [28, с. 4].

В более широком смысле СКЗИ можно назвать набором аппаратных, программных или программно-аппаратных средств, обеспечивающих одну из следующих функций [30, с. 1]:

- шифрование и расшифрование данных;

- вычисление имитовставки;
- изготовление ключевых документов;
- формирование и проверка электронной подписи;
- кодирование и декодирование данных.

1.1.2. Виртуальная частная сеть

Понятие виртуальной частной сети определяется в ГОСТ IEC 60050-732-2017 ”Международный электротехнический словарь. Часть 732. Технологии компьютерных сетей”. Согласно этому стандарту, «виртуальная частная сеть (VPN) — это компьютерная сеть, в которой для передачи данных используют промежуточные сети (например, Интернет), прозрачные для пользователей и не вводящие ограничения на протоколы, в результате чего сеть функционирует как локальная компьютерная сеть» [22, с. 2]. Примечание к стандарту указывает, что при передаче данных обычно используется туннелирование.

Говоря о VPN в контексте информационной безопасности, данное определение следует дополнить. Основной задачей VPN является не просто объединение сетей, но также обеспечение защищенного обмена информацией между ними. Фактически VPN выступает в роли виртуальной сети, которая создает защищенный канал (туннель) между участниками такого обмена.

Таким образом, более точным определением для VPN будет являться: VPN — обобщённое название технологий, позволяющих обеспечить одно или несколько сетевых соединений поверх другой сети (как правило, публичной), требующих аутентификации клиентов и использующих шифрование для обеспечения конфиденциальности и целостности передаваемых данных.

В зависимости от применяемых протоколов и назначения, можно выделить три вида VPN:

- **Узел-узел (Host-to-Host)**: данный тип соединения устанавливает защищенный туннель между двумя отдельными компьютерами в сети. Такой

тип соединения позволяет обеспечить зашифрованный обмен информацией в рамках недоверенной сетевой среды, ограничивая при этом доступ только участвующим в обмене компьютерам;

- **Узел-сеть (Remote Access VPN)**: этот тип соединения предназначен для обеспечения безопасного удаленного доступа. Он позволяет отдельным пользователям, находящимся вне корпоративной сети, безопасно подключаться к ресурсам компании, используя при этом публичные сети;

- **Сеть-сеть (Site-to-Site VPN)**: последний тип VPN используется для безопасного объединения нескольких территориально распределенных локальных сетей в единую корпоративную сеть через Интернет. VPN-шлюзы, установленные на границе каждой из локальных сетей, создают между собой постоянный зашифрованный туннель, через который проходит весь трафик.

1.1.3. Сетевой адаптер

Сетевой адаптер — это физическое аппаратное устройство, которое обеспечивает непосредственное подключение компьютера к сетевой среде.

1.1.4. Сетевой интерфейс

Сетевой интерфейс — это логическая сущность в ядре операционной системы. Именно в интерфейсе присваиваются IP-адреса и через них приложения отправляют и получают сетевые пакеты.

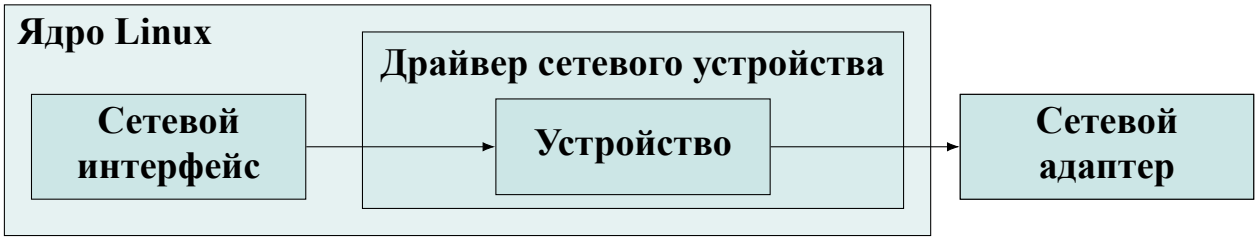

Рис. 1: Сетевой интерфейс в Linux.

1.1.5. Виртуальный сетевой интерфейс

В ГОСТ отсутствует прямое определение для термина «виртуальный сетевой интерфейс». Однако его можно вывести, опираясь на техническую документацию ядра `Linux`.

Виртуальный сетевой интерфейс — это программный компонент, который имитирует работу физического сетевого адаптера (сетевой карты). В отличие от физического интерфейса, который представляет собой физическое устройство, подключаемое к сети, виртуальный интерфейс существует исключительно на программном уровне. Операционная система и приложения взаимодействуют с виртуальным интерфейсом так же, как и с физическим.

1.1.6. Виртуальный криптографический сетевой интерфейс

Исходя из трех вышеописанных понятий, можно сформулировать комплексное определение.

Виртуальный криптографический сетевой интерфейс — это виртуальный сетевой интерфейс, который на программном уровне реализует функции средства криптографической защиты информации для обеспечения безопасной передачи данных, как правило, в рамках виртуальной частной сети (VPN).

1.1.7. Эталонная модель взаимодействия открытых систем

Эталонная модель ВОС (OSI) — это теоретическая модель, разработанная Международной организацией по стандартизации (ISO) [5, с. 28], регламентированная государственным стандартом ГОСТ Р ИСО/МЭК 7498-1-99[21, с. 3], созданная для обеспечения общей основы скоординированной разработки стандартов в области обмена информацией. Модель описывает функциональную среду ВОС и вводит концепцию семиуровневой архитектуры, где каждый уровень предоставляет услуги смежному верхнему уровню, используя функции смежного нижнего уровня:

- **Прикладной уровень (Application Layer)**: является верхним уровнем модели и предоставляет прикладным процессам средства доступа к

функциональной среде ВОС. Данный уровень обеспечивает все услуги, непосредственно воспринимаемые прикладными процессами и необходимые для обмена информацией;

- **Представления уровень (Presentation Layer)**: устанавливает способы представления информации, передаваемой между прикладными объектами. Основная задача уровня — гарантировать синтаксическую независимость прикладных объектов путем выбора общего синтаксиса передачи и выполнения необходимых преобразований;

- **Сеансовый уровень (Session Layer)**: предоставляет средства, необходимые для организации и синхронизации диалога между взаимодействующими объектами уровня представления, а также для административного управления обменом данными;

- **Транспортный уровень (Transport Layer)**: обеспечивает надежную и экономичную передачу данных между логическими объектами сеансового уровня в межоконечном режиме. Уровень оптимизирует использование доступных услуг сетевого уровня и обеспечивает требуемое качество услуг;

- **Сетевой уровень (Network Layer)**: предоставляет средства для установления, поддержания и разъединения соединений между открытыми системами, а также обеспечивает маршрутизацию и ретрансляцию данных через одну или несколько подсетей. Уровень скрывает от транспортного уровня детали использования нижележащих ресурсов передачи;

- **Канальный уровень (Data Link Layer)**: предоставляет функциональные и процедурные средства для установления, поддержания и освобождения соединений звена данных, а также для обмена блоками данных (СБД) между объектами сетевого уровня. Важной функцией является обнаружение и, по возможности, исправление ошибок, возникающих на физическом уровне;

- **Физический уровень (Physical Layer)**: обеспечивает механические, электрические, функциональные и процедурные средства для активи-

зации, поддержки и деактивизации физических соединений, предназначенных для побитовой передачи данных через физическую среду между объектами уровня звена данных.

Рис. 2: Схема семиуровневой модели OSI

1.1.8. Модель TCP/IP

Модель TCP/IP — это практическая четырехуровневая модель стека протоколов, на основе которой функционирует современный Интернет. [6; 29] В отличие от теоретической модели OSI, TCP/IP описывает реальный набор протоколов, обеспечивающих передачу данных между устройствами в глобальных сетях.

Данная модель регламентирует обмен данными и управление в сетях передачи данных и включает следующие уровни:

- **Прикладной уровень (Application Layer)**: объединяет функции прикладного и сеансового уровня модели `OSI`. На этом уровне осуществляется обмен данными между программными компонентами, реализуются специфические протоколы управления и мониторинга оборудования (например, `HTTP`, `SMTP`, `DNS`);

- **Транспортный уровень (Transport Layer)**: обеспечивает обмен данными между узлами сети, связанными непосредственно или через несколько сетей. Согласно стандарту, на этом уровне применяются два основных протокола `TCP` и `UDP`;
- **Сетевой уровень (Internet Layer)**: отвечает за межсетевое взаимодействие и маршрутизацию пакетов. Идентификация узлов в пределах сети осуществляется с помощью уникальных IP-адресов. Уровень обеспечивает доступ к информации о станции непосредственно или косвенно через шлюзы;
- **Канальный уровень (Link Layer)**: объединяет функции физического и канального уровней. Стандарт определяет использование интерфейсов локальной сети на основе протокола CSMA/CD, в частности спецификации Ethernet для физического соединения компонентов.

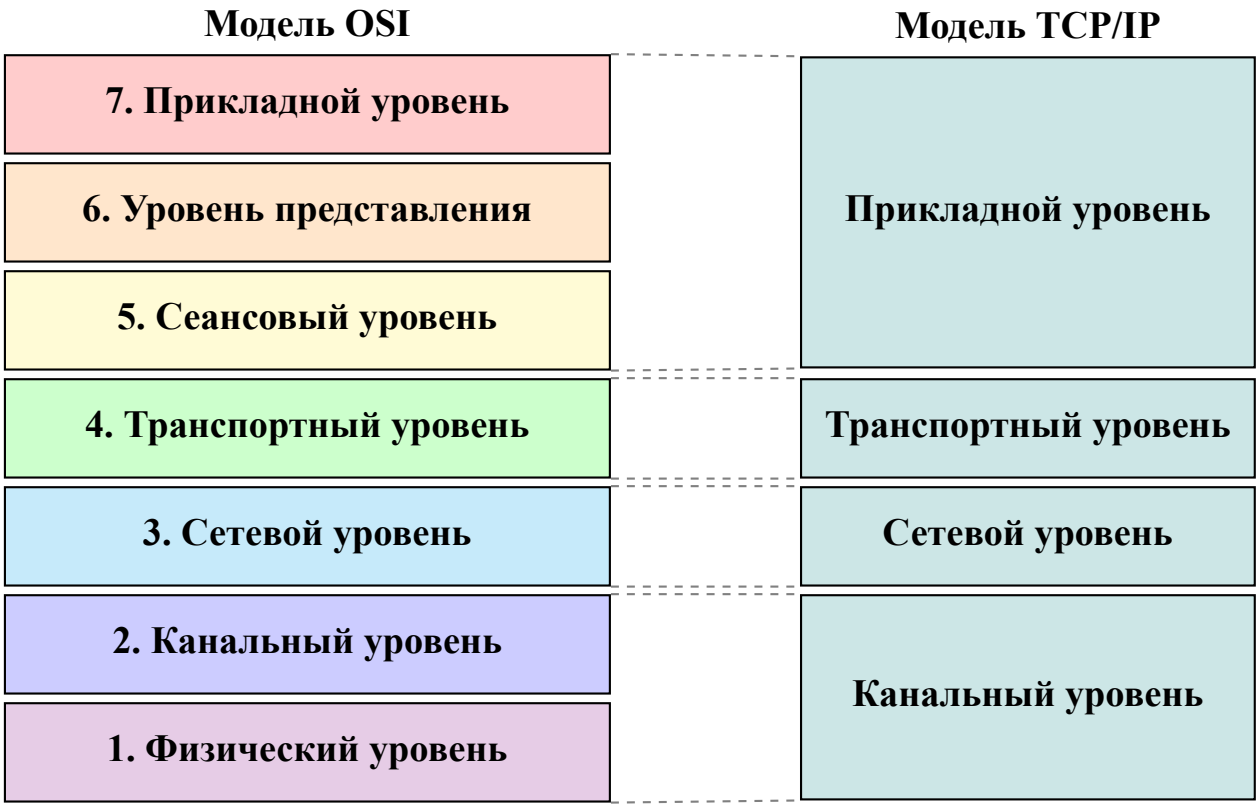

Рис. 3: Сравнение моделей OSI и TCP/IP

1.2 Теоретические основы

1.2.1. Сетевой стек операционной системы Linux

Для понимания принципов работы виртуального сетевого интерфейса `Linux` требуется сначала разобраться в сетевой подсистеме операционной системы. Сетевой стек ОС Linux представляет собой модульную иерархическую структуру, реализующую семейство протоколов TCP/IP.

Архитектурно сетевая подсистема разделена на два уровня привилегий:

1. **Пространство пользователя (User Space)**: на данном уровне располагаются прикладные приложения, которые работают с данными по средствам стандартных системных вызовов (`Socket API`);

2. **Пространство ядра (Kernel Space)**: на уровне ядра, в свою очередь, реализуются основные механизмы транспортного, сетевого и канального уровней, а также взаимодействие с аппаратными драйверами устройств.

Основной структурой данных, используемой ядром Linux для управления сетевыми пакетами, является буфер сокета (`sk_buff`). Данная структура позволяет работать с данными на различных уровнях стека без лишнего копирования их в память.

Механизм обработки и передачи данных

Процесс прохождения данных через сетевой стек базируется на двух фундаментальных операциях, обеспечивающих абстракцию уровней друг от друга:

- **Инкапсуляция (Encapsulation)**: процесс, инициируемый при передаче данных от прикладного уровня к физическому. По мере прохождения через стек, каждый уровень обрабатывает данные, полученные от вышестоящего уровня как полезную нагрузку, и добавляет к ним собственный заголовок (и концевик на канальном уровне). Эти заголовки содержат управляющую информацию (адреса, контрольные суммы,

флаги), необходимую для корректной доставки и обработки пакета на соответствующем уровне получателя;

- **Декапсуляция (Decapsulation)**: обратный процесс, происходящий при приеме данных. Сетевой интерфейс передает полученный кадр в ядро, где происходит последовательный анализ и удаление служебных заголовков. На каждом этапе стек определяет протокол вышестоящего уровня и передает ему очищенную полезную нагрузку (payload), пока исходные данные не достигнут приложения в пространстве пользователя.

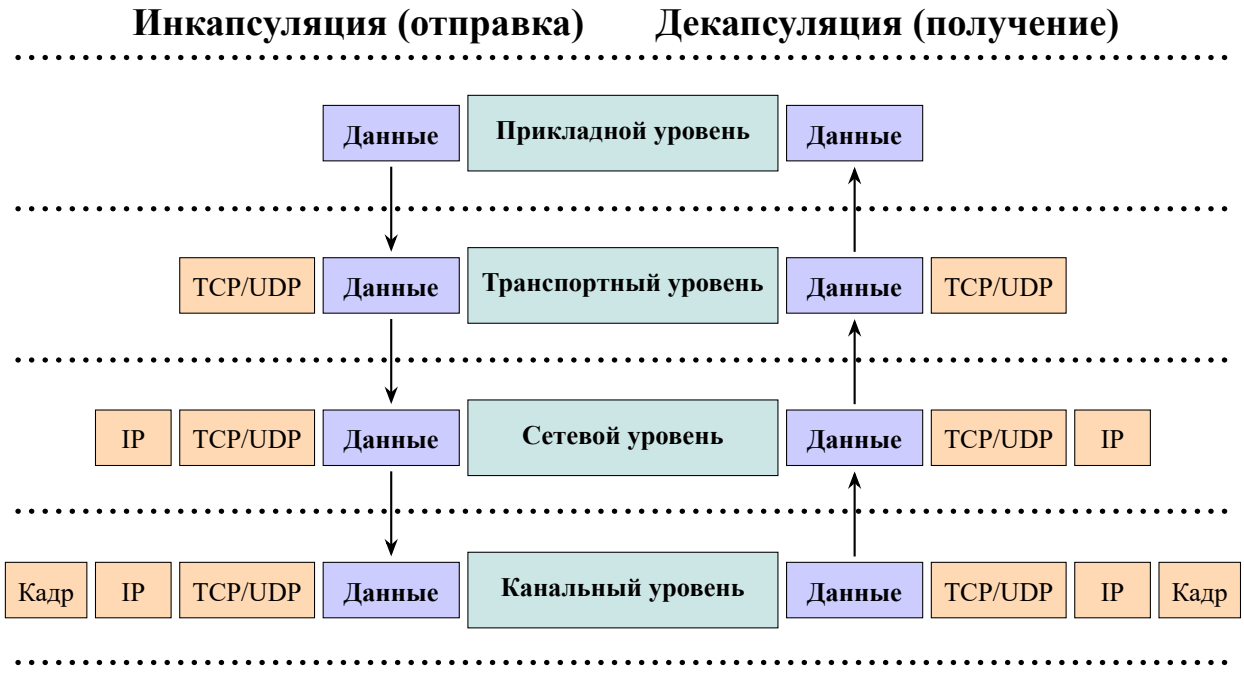

Рис. 4: Процессы инкапсуляции и декапсуляции данных в модели TCP/IP

1.2.2. Краткий анализ принципов работы WireGuard

`WireGuard` представляет собой протокол организации защищенных сетевых туннелей (*secure network tunnel*), работающий на сетевом уровне (`Layer 3`) модели `OSI`. `WireGuard` осуществляет инкапсуляцию IP-пакетов для передачи поверх транспортного протокола UDP, при этом он сильно снижает поверхность для атаки за счет своего крайне малого размера и использования фиксированного набора современных криптографических примитивов. [3]

Сам по себе `WireGuard` является довольно уникальным протоколом, в первую очередь, из-за некоторых ключевых концепций, которые заложены в его архитектуру:

Одноранговая архитектура и идентификация

`WireGuard` отходит от традиционной модели «клиент-сервер». В отличие от стандартного VPN протокола, в его архитектуре все участники сети являются равноправными узлами (так называемыми пирами). При этом идентификация каждого узла проходит по публичным ключам, которые генерируются на основе эллиптической кривой `Curve25519`.

Протокол реализует свойство «молчаливости» (*silence property*): узел не отправляет ответы на неаутентифицированные пакеты. Если входящий пакет не расшифровывается корректно известным ключом, он отбрасывается без уведомления отправителя. Это делает узлы невидимыми для сетевого сканирования.

Протокол `Noise` и установление сессии

Процедура рукопожатия (*Handshake*) построена на базе фреймворка **Noise Protocol Framework** (шаблон `IKpsk2`). Такой механизм позволяет обеспечить:

- **Взаимную аутентификацию**: идентификация узлов производится путем перекрестной проверки их статических открытых ключей (`Curve25519`) в процессе обмена сообщениями;
- **Совершенную прямую секретность (PFS):** для каждой сессии, при помощи протокола обмена ключами Диффи-Хеллмана, генерируются эфемерные ключи, которые уничтожаются сразу после завершения рукопожатия;
- **Минимизацию задержек:** установление соединения занимает всего 1.5 RTT (Round Trip Time). Инициатор может отправлять зашифрованные данные сразу после первого сообщения-рукопожатия.

Криптографическая маршрутизация (Cryptokey Routing)

В основе логики обработки трафика лежит жесткая привязка публичного ключа пира к списку разрешенных IP-адресов (`AllowedIPs`). Этот механизм объединяет функции таблицы маршрутизации и межсетевого экрана (`ACL`):

- **Исходящий трафик:** при отправке пакета `WireGuard` выбирает пира, в чьем списке `AllowedIPs` содержится IP-адрес назначения, и шифрует пакет соответствующим публичным ключом;
- **Входящий трафик:** после расшифровки пакета проверяется, соответствует ли IP-адрес источника (внутри туннеля) списку `AllowedIPs`, привязанному к ключу отправителя. При несовпадении пакет отбрасывается, что предотвращает IP-спуфинг.

Механизмы защиты

Протокол включает встроенные меры противодействия распространенным сетевым атакам:

- **Защита от DoS (Cookie):** для предотвращения истощения `CPU` дорогостоящими операциями над эллиптическими кривыми используется механизм возвратных cookie. Под нагрузкой узел может потребовать от инициатора повторить запрос с включением криптографического токена, подтверждающего владение IP-адресом, без создания состояния на сервере;
- **Защита от повторного воспроизведения (Replay Attack):** используется система монотонно возрастающих счетчиков (nonce) и скользящего окна на стороне получателя;
- **Отсутствие криптографической гибкости:** `WireGuard` использует безальтернативный набор примитивов (`ChaCha20`, `Poly1305`, `BLAKE2s`, `Curve25519`). Это исключает класс атак на понижение безопасности (*downgrade attacks*) и упрощает реализацию.

1.3 Модули ядра Linux

Основным механизмом расширения функциональности ядра Linux, не требующим его повторной сборки и перезагрузки системы, является использование загружаемых модулей ядра (Loadable Kernel Modules, LKM). [12]

Данный механизм позволяет динамически загружать и выгружать программный код в пространство ядра работающей операционной системы. Это обеспечивает модульность и гибкость архитектуры: драйверы устройств, файловые системы и реализации сетевых протоколов могут существовать в виде отдельных компонентов, подключаемых только по мере необходимости.

С архитектурной точки зрения модуль представляет собой специализированный объектный файл, который связывается (линкуется) с ядром во время выполнения. Жизненный цикл любого модуля определяется двумя ключевыми этапами:

- **Инициализация**: процесс, выполняемый при загрузке модуля в память. На этом этапе модуль регистрирует свои функции в соответствующих подсистемах ядра и выделяет необходимые ресурсы;
- **Деинициализация**: процесс, выполняемый при выгрузке модуля. Он отвечает за корректное освобождение ресурсов и удаление регистрации, возвращая систему в исходное состояние.

Именно этот архитектурный подход используется для расширения криптографической подсистемы: новые алгоритмы шифрования и хэширования оформляются в виде модулей, которые регистрируют свои реализации через интерфейсы Crypto API.

1.4 Криптографическая подсистема ядра Linux

Криптографическая подсистема ядра Linux (Linux Kernel Crypto API) представляет собой унифицированный программный интерфейс, предоставляющий доступ к криптографическим примитивам для других подсистем

ядра (например, сетевого стека или дискового шифрования). [9]

Архитектура фреймворка позволяет абстрагироваться от конкретной реализации алгоритма, прозрачно поддерживая как программные, так и аппаратно-ускоренные решения. Реализация любого криптографического алгоритма в рамках этой подсистемы рассматривается как «преобразование» (transformation), которое регистрируется в ядре и становится доступным для использования через стандартные вызовы API.

1.4.1. Классификация криптографических примитивов

Архитектура `Crypto API` классифицирует алгоритмы не только по их функциональному назначению, но и по способу взаимодействия с памятью. Каждому классу соответствует свой набор абстракций и структур данных.

Ниже приведен перечень основных типов алгоритмов, поддерживаемых подсистемой:

Симметричные шифры (SKCIPHER)

Аббревиатура расшифровывается как *Symmetric Key Cipher*. Это основной класс алгоритмов для шифрования данных произвольной длины.

- **Назначение:** реализация режимов работы блочных шифров (CBC, CTR, XTS) или потоковых шифров;
- **Особенности:** ключевой особенностью данного API является работа с данными через механизм `scatterlist`. Это позволяет обрабатывать данные, фрагментированные в физической памяти (например, сетевые пакеты, части которых располагаются в разных страницах памяти), без их предварительного копирования в непрерывный буфер.

2. Базовые блочные шифры (CIPHER)

Низкоуровневый интерфейс, реализующий преобразование ровно одного блока фиксированной длины (например, 8 байт для ГОСТ 28147-89 или 16 байт для AES).

- **Назначение:** используется преимущественно как строительный блок для создания более сложных конструкций (шаблонов), таких как `skcipher` или хэш-функции. Напрямую прикладными модулями используется редко;
- **Особенности:** работает синхронно и только с прямыми адресами памяти, не поддерживая `scatterlist`.

Аутентифицированное шифрование (AEAD)

Класс *Authenticated Encryption with Associated Data* объединяет операции шифрования и вычисления имитовставки (`MAC`) в один атомарный вызов.

- **Назначение:** защита сетевого трафика в современных VPN-протоколах (`WireGuard`, `IPsec`). Гарантирует целостность не только зашифрованной полезной нагрузки, но и открытых заголовков пакета;
- **Особенности:** принимает на вход два потока данных: ассоциированные данные (которые не шифруются, но защищаются от подделки) и данные для шифрования.

Протоколы согласования ключей (KPP)

Аббревиатура расшифровывается как *Key-agreement Protocol Primitives*. Этот класс абстрагирует операции асимметричной криптографии, необходимые для выработки общего секрета.

- **Назначение:** реализация механизмов рукопожатия (handshake), таких как `Diffie-Hellman` (`DH`), `ECDH` и их аналогов, включая `VKO` (ГОСТ Р 34.10-2012);
- **Принцип работы:** реализация предоставляет три основные операции: загрузка собственного секретного ключа, генерация соответствующего публичного ключа и вычисление общего секрета на основе принятого публичного ключа удаленной стороны.

Хэш-функции (SHASH / AHASH)

Интерфейс для вычисления дайджестов сообщений и кодов аутентификации (`HMAC`). В ядре разделен на два подтипа:

- **SHASH (Synchronous Hash):** оптимизирован для программных реализаций. Операции выполняются в контексте вызывающего процесса без переключений контекста;
- **AHASH (Asynchronous Hash):** предназначен для аппаратных ускорителей или обработки больших объемов данных. Может возвращать управление до завершения вычисления, используя механизм callback-функций.

Асимметричные шифры (AKCIPHER)

Класс *Asymmetric Key Cipher* предназначен для алгоритмов с публичным ключом общего назначения.

- **Назначение:** реализация алгоритмов `RSA`, создание и проверка электронных цифровых подписей (`DSA`, `ECDSA`), а также шифрование небольших блоков данных асимметричным методом.

Генераторы случайных чисел (RNG)

Обеспечивает унифицированный интерфейс к источникам энтропии и детерминированным генераторам псевдослучайных чисел (`DRBG`), необходимых для генерации ключей, векторов инициализации и одноразовых кодов.

1.4.2. Протокол `Netlink`

Для управления виртуальным интерфейсом и мониторинга его состояния необходим надежный канал связи между пространством ядра (`kernelspace`) и пространством пользователя (`userspace`). В современной архитектуре Linux стандартом для таких задач является протокол `Netlink`. [11]

Это механизм межпроцессного взаимодействия (`IPC`), который эмулирует сетевое взаимодействие: обмен данными происходит через сокеты, что позволяет использовать стандартные системные вызовы для отправки и получения конфигурационных команд. В отличие от устаревшего системного вызова `ioctl`, `Netlink` является асинхронным и событийно-ориентированным протоколом.

Архитектура обмена данными

Взаимодействие строится на обмене пакетированными сообщениями. Протокол поддерживает две основные модели коммуникации:

- **Синхронные запросы (Unicast):** приложение отправляет команду (например, «добавить пира» или «получить список ключей») и ожидает от ядра подтверждения или запрошенных данных;
- **Асинхронные уведомления (Multicast):** ядро может самостоятельно инициировать передачу данных группе подписчиков. Это критически важно для VPN-интерфейсов, так как позволяет мгновенно уведомлять приложения об изменении состояния соединения (например, о завершении рукопожатия) без необходимости постоянного опроса (polling).

Гибкость протокола обеспечивается благодаря тому, что данные передаются в формате `TLV` (`Type-Length-Value`). Вместо заданных в исходном коде структур языка «`C`», сообщения формируются как набор атрибутов. Это позволяет добавлять новые параметры конфигурации в будущих версиях модуля, не нарушая работу старых утилит управления (обратная совместимость).

Generic Netlink

Классический `Netlink` имеет ограничение на количество поддерживаемых протоколов. Для решения этой проблемы был разработан механизм `Generic Netlink`.

Он выступает в роли мультиплексора, позволяя загружаемым модулям ядра (таким как `WireGuard`) динамически регистрировать собственные ”семейства” команд. При регистрации модуль получает уникальный идентификатор, по которому утилиты из пространства пользователя могут к нему обращаться. Это избавляет разработчика от необходимости резервировать статические номера протоколов в исходном коде ядра.

2 Аналитическая часть

2.1 Разработка и эксплуатация модулей ядра

Для разработки модулей, реализующих необходимые криптографические протоколы, в первую очередь, необходимо разобраться в том, как эти самые модули работают в контексте ядра. Для этого нужно рассмотреть их структуру, жизненный цикл, процесс сборки и методы их отладки.

2.1.1. Структура и программная реализация

Любой модуль ядра должен придерживаться определенного соглашения о том, как он должен выглядеть с точки зрения реализации. В отличие от стандартного приложения написанного на языке «C», в модулях отсутствует стандартная точка входа в виде функции `main()`, вместо этого в модулях используются специальные макросы, которые позволяют реализовать точки входа и выхода из модуля.

Для того чтобы модуль считался валидным, он должен содержать как минимум следующее:

- **Заголовочные файлы ядра**: обеспечивают доступ к внутренним структурам и функциям API ядра (например, `<linux/module.h>`, `<linux/kernel.h>`);
- **Макросы инициализации**: `module_init()` указывает ядру, какую функцию вызвать при загрузке;
- **Макросы завершения**: `module_exit()` указывает функцию для очистки ресурсов при выгрузке;
- **Метаданные**: макросы `MODULE_LICENSE`, `MODULE_AUTHOR` и другие, описывающие модуль для утилит управления.

Ниже представлен минимальный пример модуля ядра на языке «C»:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/printk.h>
```

```c
#include <linux/init.h>

static int __init module_init(void) {
    printk(KERN_INFO "Module loaded.\n");
    return 0;
}

static void __exit module_exit(void) {
    printk(KERN_INFO "Module unloaded.\n");
}

module_init(module_init);
module_exit(module_exit);
MODULE_LICENSE("GPL");
```

[12]

2.1.2. Система сборки Kbuild

Для сборки модулей ядра требуется полная бинарная совместимость символов и структур данных с ядром, для которого собирается модуль, именно поэтому сборка модулей происходит в контексте исходного кода ядра необходимой версии. Для этого используется система сборки «`Kbuild`». Обычно для автоматизации процесса сборки используется скрипт `Makefile`, в котором автоматически указывая путь к заголовочным файлам (`linux-headers`) и список необходимых объектных файлов.

Примерно так будет выглядеть минимальный `Makefile` для сборки модуля ядра:

```
obj-m += my_module.o

PWD := $(CURDIR)

all:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

[10]

Результатом успешной сборки будет объектный файл модуля с расширением (`.ko – kernel object`). Такой объект может быть загружен в ядро `Linux`.

2.1.3. Управление загрузкой

Для работы с загружаемыми модулями в ОС `Linux` предусмотрены специальные утилиты:

- `insmod <путь_к_файлу.ko>`: загружает модуль в ядро. При этом выполняется его `init` функция;
- `rmmod <имя_модуля>`: выгружает модуль из ядра. При этом выполняется `exit` функция;
- `lsmod`: показывает список всех загруженных на данный момент модулей;
- `modprobe`: более высокоуровневая утилита, которая умеет автоматически загружать модуль вместе с его зависимостями, используя базу данных модулей.

Информацию о загруженных модулях также можно посмотреть в `/proc/modules` и `/sys/module`.

2.1.4. Методы отладки в пространстве ядра

Отладка кода, работающего в пространстве ядра, является намного более сложной задачей, нежели отладка обычного приложения. В контексте ядра любая критическая ошибка (например, разыменование `NULL`-указателя) может привести к полной остановке системы — состоянию `Kernel Panic`. Стандартные отладчики (`GDB`) здесь применимы только при использовании виртуальных машин или аппаратных `JTAG`-интерфейсов.

- **Отладка с помощью `printk`**: самый простой и распространенный способ — вставка отладочных сообщений в код. `printk` позволяет указывать уровень важности сообщения (например, `KERN_INFO`,

KERN_DEBUG). Уровнем вывода сообщений на консоль можно управлять через /proc/sys/kernel/printk. Для удобства трассировки часто используются предопределенные макросы __FILE__, __LINE__ и __func__;

- **Анализ Kernel Oops**: "Oops" — это сообщение о некритической ошибке ядра. В отличие от "Panic" система пытается продолжить работу, но её состояние уже нестабильно. Такое сообщение содержит в себе ключевую информацию для отладки:
  - причина ошибки (BUG: unable to handle kernel paging request);
  - адрес инструкции, вызвавшей сбой, и смещение внутри функции (EIP is at my_oops_init+0x5/0x20);
  - код ошибки, который описывает тип операции (чтение/запись, режим ядра/пользователя);
  - состояние регистров и трассировку стека вызовов (Call Trace).
- **Использование objdump и addr2line**: имея адрес инструкции из сообщения "Oops" и базовый адрес загрузки модуля (из /proc/modules), можно точно определить строку в исходном коде, которая привела к сбою.
  - objdump: дизассемблирует код модуля, сопоставляя его с исходным кодом;

    ```
    objdump -dS --adjust-vma=<базовый_адрес> module.ko
    ```

  - addr2line: преобразует смещение адреса внутри функции в номер строки исходного кода.

    ```
    addr2line -e module.o <смещение>
    ```

- **Динамическая отладка (dyndbg)**: более гибкая методика, позволяющая включать и выключать отладочные сообщения (pr_debug()) на лету, без повторной сборки модуля. Для этого ядро должно быть собрано с опцией CONFIG_DYNAMIC_DEBUG. Управление осуществляется через запись в файл /sys/kernel/debug/dynamic_debug/control, поз-

воляя фильтровать сообщения по имени файла, функции, номеру строки или формату сообщения;

- **Отладчик ядра (`KDB/KGDB`)**: для сложной отладки можно использовать встроенный отладчик ядра `KDB`. Он позволяет останавливать выполнение ядра, устанавливать точки останова (`breakpoints`), просматривать и изменять содержимое памяти и регистров в «живой» системе, обычно через последовательный порт.

2.2 Архитектура и регистрация криптографических модулей

Разработка собственного криптографического модуля в ядре `Linux` является довольно тривиальной задачей. В ядре существует базовая структура, которая описывает базовые свойства алгоритма, на основе которой уже строятся структуры алгоритмов необходимого типа.

2.2.1. Структура дескриптора алгоритма

Базовым строительным блоком подсистемы является структура `struct crypto_alg`. Она содержит метаданные, необходимые для того, чтобы ядро смогло идентифицировать алгоритм и в дальнейшем выбрать наилучшую реализацию и управлять памятью. [13]

```
struct crypto_alg {
    struct list_head cra_list;
    struct list_head cra_users;

    u32 cra_flags; /* Набор флагов, описывающих алгоритм */

    unsigned int cra_blocksize; /* Размер блока шифра в байтах */
    unsigned int cra_ctxsize;   /* Размер структуры контекста */
    unsigned int cra_alignmask; /* Маска выравнивания входных
                                   и выходных данных */

    int cra_priority; /* Приоритет реализации */

    refcount_t cra_refcnt;

    char cra_name[CRYPTO_MAX_ALG_NAME];         /* Имя алгоритма */
```

```
    char cra_driver_name[CRYPTO_MAX_ALG_NAME]; /* Имя драйвера */

    const struct crypto_type *cra_type;  /* Тип криптопреобразования */
    /* Указатель на структуру crypto_type, реализующую
     * общие функции обратного вызова.
     *
     * Доступные варианты:
     * - &crypto_blkcipher_type
     * - &crypto_ablkcipher_type
     * - &crypto_ahash_type
     * - &crypto_rng_type
     *
     * Оставляется пустым для шифрования блока (cipher),
     * сжатия (compress) и блокирующего хэширования (shash).
     */

    union {
        struct cipher_alg cipher;
        struct compress_alg compress;
    } cra_u;

    int (*cra_init)(struct crypto_tfm *tfm);
    void (*cra_exit)(struct crypto_tfm *tfm);
    void (*cra_destroy)(struct crypto_alg *alg);

    struct module *cra_module;
};
```

2.2.2. Классификация криптографических примитивов

Архитектура `Crypto API` классифицирует алгоритмы не только по их функциональному назначению, но и по способу взаимодействия с памятью. Каждому классу соответствует свой набор абстракций и структур данных.

Основные типы алгоритмов

- **CIPHER (Базовые блочные шифры):** низкоуровневый синхронный интерфейс для преобразования одного блока данных. Используется как строительный элемент для более сложных режимов;

- **SKCIPHER (Симметричные шифры):** основной API для шифрования данных произвольной длины (потоков). Работает с фрагментированной памятью (scatterlist) и поддерживает векторы инициализации, что делает его пригодным для обработки сетевых пакетов;

- **AKCIPHER (Асимметричные шифры):** API для алгоритмов с открытым ключом, таких как RSA, выполняющих операции шифрования/расшифрования небольших объемов данных (например, других ключей);

- **AEAD (Аутентифицированное шифрование):** ключевой тип для современных защищенных протоколов. Объединяет шифрование и вычисление имитовставки (MAC) в одну атомарную операцию. Это позволяет гарантировать не только конфиденциальность, но и целостность/-подлинность данных (например, связки AES-GCM или «Кузнечик-MGM»);

- **HASH (SHASH / AHASH):** интерфейсы для вычисления хэш-сумм и кодов аутентификации (HMAC). Разделены на синхронный (shash) — для быстрых программных реализаций, и асинхронный (ahash) — для аппаратных ускорителей;

- **KPP (Протоколы согласования ключей):** API для реализации механизмов асимметричной криптографии, используемых для выработки общего секрета, например, Diffie-Hellman, ECDH и VKO. Это основа для этапа рукопожатия (handshake);

- **SIG (Цифровые подписи):** специализированный API для создания и проверки цифровых подписей (например, ECDSA, RSA-PSS). В отличие от AKCIPHER, фокусируется на операциях sign и verify;

- **COMPRESSION (SCOMP/ACOMP):** интерфейсы для синхронного (scomp) и асинхронного (acomp) сжатия данных;

- **RNG (Генераторы случайных чисел):** предоставляет доступ к источникам энтропии для генерации ключей, векторов инициализации и других криптографических материалов.

В данной работе нас интересуют три следующих типа алгоритмов: hash, aead и kpp.

Хэш-функции (SHASH / AHASH)

Данный тип алгоритма `Crypto API` позволяет реализовать алгоритм и процедуру вычисления хэш-функции (в данном случае, ГОСТ Р 34.11-2012 «Стрибог») [25], а также кодов аутентификации сообщений (`HMAC`). Работа с `API` строится на итеративной обработке данных и управлении внутренним состоянием алгоритма. Для реализации подобного алгоритма необходимо реализовать набор функций-обработчиков, описывающих жизненный цикл вычисления:

- **`init`**: инициализирует контекст хэш-функции, загружая в него начальные значения, стандартизованные для данного алгоритма;
- **`update`**: принимает на вход очередной фрагмент данных и обновляет внутреннее состояние. Может вызываться многократно для обработки больших сообщений;
- **`final`**: завершает вычисление. Выполняет финальное дополнение (padding) сообщения, производит последнюю итерацию функции сжатия и записывает итоговый дайджест в выходной буфер.

Для регистрации такого алгоритма необходимо заполнить структуру `shash_alg`. [14]

```
#include <crypto/shash.h>

static int hash_init(struct shash_desc desc) { /* */ return 0; }
static int hash_update(struct shash_desc *desc,
    const u8 data, unsigned int len) { /* */ return 0; }
static int hash_final(struct shash_desc *desc, u8 out) { /* */ return 0; }

static struct shash_alg my_alg = {
    .init = hash_init,
    .update = hash_update,
    .final = hash_final,
    .digestsize = 32,
    .descsize = sizeof(struct hash_ctx),
    .base = {
        .cra_name = "hash",
```

```
        .cra_driver_name= "hash-generic",
        .cra_blocksize = 64,
        .cra_module = THIS_MODULE,
    }
};

static int __init hash_mod_init(void)
{
    return crypto_register_shash(&my_alg);
}

static void __exit hash_mod_exit(void)
{
    return crypto_unregister_shash(&my_alg);
}
```

Аутентифицированное шифрование (AEAD)

Данный тип алгоритма `Crypto API` предназначен для реализации процедур аутентифицированного шифрования (в данном случае, «Кузнечик» в режиме `MGM`). `API` этого типа атомарно обеспечивает конфиденциальность и целостность, что является стандартом для защиты сетевого трафика. Для реализации подобного алгоритма необходимо реализовать две основные функции:

- **`encrypt`**: получает на вход открытый текст, вектор инициализации (IV) и ассоциированные данные (заголовки, которые не шифруются, но защищаются от подмены). На выходе формирует шифротекст и тег аутентификации (имитовставку);
- **`decrypt`**: выполняет обратную операцию. Функция принимает шифротекст, IV, ассоциированные данные и тег. Она самостоятельно вычисляет тег для полученных данных и сравнивает его с уже переданным. Только в случае полного совпадения расшифрованный текст считается валидным, иначе возвращается ошибка, предотвращая обработку поддельных пакетов.

Регистрация AEAD-алгоритма производится через структуру `aead_alg`. [15]

```c
#include <crypto/aead.h>

static int aead_setkey(struct crypto_aead *tfm, const u8 key,
unsigned int keylen) { /* */ return 0; }
static int aead_encrypt(struct aead_request req) { /* */ return 0; }
static int aead_decrypt(struct aead_request req) { /* */ return 0; }

static struct aead_alg alg = {
    .setkey = aead_setkey,
    .encrypt = aead_encrypt,
    .decrypt = aead_decrypt,
    .ivsize = 16,
    .maxauthsize = 16,
    .base = {
        .cra_name = "aead",
        .cra_driver_name= "aead-generic",
        .cra_blocksize = 1,
        .cra_ctxsize = sizeof(struct aead_ctx),
        .cra_module = THIS_MODULE,
    }
};

static int __init aead_mod_init(void)
{
return crypto_register_aead(&alg);
}

static void __exit aead_mod_exit(void)
{
crypto_unregister_aead(&alg);
}
```

Протоколы согласования ключей (KPP)

Данный тип алгоритма `Crypto API` реализует асимметричные примитивы, необходимые для выработки общего секрета (в данном случае это VKO на базе ГОСТ Р 34.10-2012). API предоставляет три основные операции:

- **`set_secret`**: загружает собственный закрытый ключ в контекст алгоритма;

- **`generate_public_key`**: вычисляет публичный ключ, соответствующий ранее загруженному закрытому ключу;
- **`compute_shared_secret`**: принимает на вход публичный ключ удаленной стороны и, используя свой закрытый ключ, вычисляет общий секрет, который в дальнейшем используется для генерации сессионных симметричных ключей.

Регистрация KPP-алгоритма производится через структуру `kpp_alg`. [16]

```
#include <crypto/kpp.h>

static int kpp_set_secret(struct crypto_kpp *tfm, const void key,
unsigned int keylen) { /* */ return 0; }
static int kpp_gen_pubkey(struct kpp_request req) { /* */ return 0; }
static int kpp_compute_ss(struct kpp_request req) { /* */ return 0; }
static unsigned int kpp_max_size(struct crypto_kpp *tfm) { return 32; }

static struct kpp_alg alg = {
    .set_secret = kpp_set_secret,
    .generate_public_key = kpp_gen_pubkey,
    .compute_shared_secret = kpp_compute_ss,
    .max_size = kpp_max_size,
    .base = {
        .cra_name = "kpp",
        .cra_driver_name= "kpp-generic",
        .cra_ctxsize = sizeof(struct kpp_ctx),
        .cra_module = THIS_MODULE,
    }
};

static int __init kpp_mod_init(void)
{
return crypto_register_kpp(&alg);
}

static void __exit kpp_mod_exit(void)
{
crypto_unregister_kpp(&alg);
}
```

2.3 Анализ криптографической архитектуры WireGuard

Главным этапом подготовки к адаптированию WireGuard на криптографические протоколы стандартизированные в ГОСТ является анализ архитектуры WireGuard на предмет того, где и какие протоколы в нем используются.

Реализация WireGuard довольно проста, и практически вся интересующая информация может быть найдена в файлах: noise.c, device.c, send.c/receive.c и cookie.c.

В центре самого WireGuard располагается noise.c. Он реализует машину состояний протокола рукопожатия на базе фреймворка Noise Protocol Framework (шаблон IKpsk2).

2.3.1. Noise Framework

В основе логики WireGuard лежит протокол Noise, который представляет собой не единый алгоритм, а конструктор протоколов. Автор WireGuard выбрал конкретную конфигурацию — **Noise_IKpsk2**. Эта строка конфигурации жестко зашита в исходном коде ядра и определяет последовательность криптографических операций. [3]

В файле noise.c эта конфигурация определяется следующим образом: [17]

```
static const u8 handshake_name[37] __nonstring =
    "Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s";
```

На основе этой строки можно примерно представить список криптографических примитивов, которые в дальнейшем необходимо будет заменить:

- **IKpsk2**: Паттерн рукопожатия. Означает, что статический ключ инициатора передается сразу (I), статический ключ респондера известен заранее (K), используется пре-распределенный ключ (psk) и обмен завершается за 2 сообщения;
- **25519**: Алгоритм Диффи-Хеллмана на кривой Curve25519;

- **ChaChaPoly**: AEAD-шифр ChaCha20-Poly1305;
- **BLAKE2s**: Хэш-функция.

Обмен ключами (Key Exchange)

Текущая реализация:

В оригинальном протоколе используется эллиптическая кривая Curve25519. В исходном коде (noise.c) ключевой функцией является mix_dh, которая выполняет умножение точки на скаляр и обновляет состояние цепочки ключей. [17]

```
static bool __must_check mix_dh(u8 chaining_key[NOISE_HASH_LEN],
                                u8 key[NOISE_SYMMETRIC_KEY_LEN],
                                const u8 private[NOISE_PUBLIC_KEY_LEN],
                                const u8 public[NOISE_PUBLIC_KEY_LEN])
{
    u8 dh_calculation[NOISE_PUBLIC_KEY_LEN];

    if (unlikely(!curve25519(dh_calculation, private, public)))
        return false;

    kdf(chaining_key, key, NULL, dh_calculation, NOISE_HASH_LEN,
        NOISE_SYMMETRIC_KEY_LEN, 0, NOISE_PUBLIC_KEY_LEN, chaining_key);
    /* ... */
}
```

Адаптация под ГОСТ:

Согласно ГОСТ Р 34.10-2012 алгоритм Curve25519 следует заменить на алгоритм VKO (выработка ключа обмена).

Алгоритм VKO (Выработка Ключа Обмена) является прямым функциональным аналогом (изоморфным примитивом) для ECDH среди протоколов стандартизированных ГОСТ. Он реализует операцию умножения точки эллиптической кривой (открытого ключа одной стороны) на скаляр (закрытый ключ другой стороны) для получения общей точки, координаты которой используются в качестве общего секрета. VKO идеально ложится в эту схему,

меняя лишь математическую основу, логика работы протокола при этом остается неизменной.

Проблемы реализации:

- **Размер ключей:** `Curve25519` использует 32-байтные ключи. В ГОСТ Р 34.10-2012 использует 64-байтные публичные ключи (в виде двух координат по 32 байта). Для корректной работы необходимо будет обновить структуры хранящие в себе информацию о пирах (`struct wg_peer`) и исправить формата пакетов рукопожатия, так как старые заголовки не вместят ключи увеличенного размера;
- **Endianness:** `WireGuard` работает с little-endian числами, тогда как в криптографии ГОСТ используется big-endian.

Симметричное шифрование (AEAD)

Текущая реализация:

Для защиты как сообщений рукопожатия, так и полезной нагрузки (IP-пакетов) используется **ChaCha20-Poly1305**. Это крайне быстрый с точки зрения производительности поточный шифр. Вызов шифрования в `send.c` выглядит так: [18]

```
/* send.c - функция encrypt_packet */
return chacha20poly1305_encrypt_sg_inplace(sg, plaintext_len, NULL, 0,
                                           PACKET_CB(skb)->nonce,
                                           keypair->sending.key);
```

Адаптация под ГОСТ: В качестве замены схеме `ChaCha20-Poly1305` была выбрана комбинация блочного шифра «Кузнечик» в режиме гаммирования с мультипликативной имитовставкой — `MGM` (`Multilinear Galois Mode`), регламентированном в рекомендациях `Р 1323565.1.026–2019`. [27]

Оригинальный `WireGuard` использует `AEAD`-схему (`Authenticated Encryption with Associated Data`). Это означает, что алгоритм должен не только шифровать данные, но и вычислять криптографически стойкую контрольную сумму (`Tag`) от зашифрованного текста и открытых заголовков

пакета (Associated Data). Режим MGM является единственным современным отечественным стандартом, который архитектурно соответствует требованиям AEAD.

Хотя «Магма» быстрее «Кузнечика» в программной реализации, она имеет размер блока 64 бита. В высоконагруженных VPN-каналах (1 Гбит/с и выше) это создает риск коллизий блока уже после передачи нескольких десятков гигабайт данных (атака Sweet32). [7] Для долгоживущих туннелей использование 64-битных блочных шифров в современной криптографии считается небезопасным. «Кузнечик» с блоком 128 бит лишен этого недостатка.

Проблемы реализации:

- **Производительность:** «Кузнечик» в отличие от ChaCha20 намного более тяжеловесный блочный шифр (SP-сеть). Без использования оптимизации в виде векторных инструкций (AVX2/AVX-512) или аппаратного ускорения пропускная способность упадет в разы по сравнению с ChaCha20;
- **Паддинг и выравнивание:** ChaCha20 в отличие от «Кузнечика» является поточным шифром, ему не важна длина данных. «Кузнечик» — блочный (128 бит), так что для корректной его работы необходимо использовать режим MGM, который позволит ему работать как поточный шифр. Это значительно усложнит его реализацию.

Хэширование и KDF

Текущая реализация:

Функция BLAKE2s используется везде: для вычисления хэша сессии (h), для деривации ключей (HKDF) и для вычисления MAC в cookie.c. [17; 19]

```
/* noise.c - обновление хэша сессии */
static void mix_hash(u8 hash[NOISE_HASH_LEN], const u8 *src, size_t src_len)
{
    struct blake2s_ctx blake;
    blake2s_init(&blake, NOISE_HASH_LEN);
    blake2s_update(&blake, hash, NOISE_HASH_LEN);
    blake2s_update(&blake, src, src_len);
```

```
    blake2s_final(&blake, hash);
}

/* cookie.c */
static void compute_mac1(...) {
    blake2s(key, NOISE_SYMMETRIC_KEY_LEN, message, len, mac1, COOKIE_LEN);
}
```

Адаптация под ГОСТ: Необходима замена на функцию «Стрибог» с длиной выхода 256 бит.

Выбор 256-битной версии «Стрибога» позволяет сохранить размерность внутренних структур данных (хэш сессии, цепочка ключей), идентичную оригиналу. Это минимизирует вмешательство в логику машины состояний `Noise`: меняется лишь математическая реализация функции сжатия, но не формат данных, передаваемых между этапами рукопожатия.

Проблемы реализации:

- **Контекст выполнения:** Хэширование в `WireGuard` часто происходит на ”горячем пути” (hot path). «Стрибог» значительно медленнее `BLAKE2s`. Это может замедлить скорость установки соединений, особенно под большой нагрузкой;
- **HMAC конструкция:** `BLAKE2s` имеет встроенный режим `keyed-hashing`. «Стрибог» требует использования классической конструкции `HMAC`, что добавляет лишние вызовы функции сжатия;
- **DDoS атаки:** «Стрибог» значительно замедляет скорость генерации MAC-контрольных сумм, проверка MAC1 происходит для каждого входящего пакета рукопожатия. Это может увеличить эффективность DDoS атак на сервер.

2.4 Обзор отечественных криптографических стандартов

Для обеспечения защиты информации в соответствии с требованиями регулятора ФСТЭК, в модифицированную версию `WireGuard` интегрируются алгоритмы, определенные в стандартах ГОСТ. Ниже приведено детальное описание их криптографической структуры.

2.4.1. Алгоритм хэширования ГОСТ Р 34.11-2012 «Стрибог»

В качестве основы для механизмов контроля целостности, защиты от DoS-атак и деривации ключей используется функция хэширования, определенная в ГОСТ Р 34.11-2012. В рамках данной работы применяется версия алгоритма с длиной хэш-кода 256 бит, что позволяет сохранить совместимость с размерностью внутренних структур протокола `WireGuard`. [25]

Алгоритм реализует функцию сжатия на основе блочного шифра с архитектурой SP-сети (Substitution-Permutation network) и использует итерационную конструкцию Меркла-Дамгарда. Все операции выполняются над векторами из пространства V_{512} (двоичные векторы длины 512 бит).

Базовые преобразования

Математический аппарат алгоритма строится на композиции четырех преобразований, определенных для векторов $a \in V_{512}$:

1. **Преобразование X (XOR)**: Покомпонентное сложение двух векторов по модулю 2. Для $k, a \in V_{512}$:

$$X[k](a) = k \oplus a$$

2. **Преобразование S (SubBytes)**: Нелинейное биективное преобразование. Вектор a разбивается на 64 байта ($a = a_{63} \| \dots \| a_0$), каждый из которых заменяется в соответствии с фиксированной перестановкой π (S-box) в кольце вычетов Z_{2^8}:

$$S(a) = \pi(a_{63}) \| \pi(a_{62}) \| \dots \| \pi(a_0)$$

3. **Преобразование P (Transposition)**: Перестановка байтов вектора для обеспечения диффузии. Используется фиксированная перестановка τ:

$$P(a) = a_{\tau(63)} \| a_{\tau(62)} \| \dots \| a_{\tau(0)}$$

4. **Преобразование L (Linear transformation)**: Линейное преобразова-

ние. Входной вектор умножается справа на фиксированную двоичную матрицу A размером 64×64 над полем $GF(2)$. Операция обеспечивает зависимость каждого бита выходного вектора от всех битов входного:

$$L(a) = A \cdot a$$

Функция сжатия

Функция сжатия $g_N(h, m, N)$ обновляет текущее состояние хэша $h \in V_{512}$ на основе очередного блока сообщения $m \in V_{512}$ и счетчика количества обработанных бит $N \in V_{512}$.

Процедура вычисления выглядит следующим образом:

1. **Формирование ключа K:** Текущее состояние хэша смешивается со счетчиком длины и подвергается преобразованиям L, P, S:

$$K = LPS(h \oplus N) = L(P(S(h \oplus N)))$$

2. **Шифрование (E):** Выполняется шифрование блока сообщения m на ключе K. Функция шифрования $E(K, m)$ состоит из 12 раундов.

   Сначала генерируются 13 раундовых ключей $K_1, \ldots, K_{13}$ следующим образом:

$$\begin{cases} K_1 = K \\ K_i = LPS(K_{i-1} \oplus C_{i-1}), \quad i = 2, \ldots, 13 \end{cases}$$

   где C_i — итерационные константы, определенные стандартом.

   Процесс преобразования состояния $State$ (изначально $State = m$): Для i от 1 до 12 выполняется раундовое преобразование:

$$State \leftarrow LPS(State \oplus K_i)$$

   Финальный шаг:

$$E(K, m) = State \oplus K_{13}$$

3. **Схема Миягучи-Пренеля:** Новое значение хэш-кода получается пу-

тем сложения по модулю 2 результата шифрования, исходного блока сообщения и предыдущего значения хэша:

$$g_N = E(K, m) \oplus h \oplus m$$

2.4.2. Блочный шифр ГОСТ Р 34.12-2015 «Кузнечик»

Для замены алгоритма `ChaCha20` в качестве примитива симметричного шифрования выбран блочный шифр «Кузнечик» (Kuznyechik), определенный в ГОСТ Р 34.12-2015. [26; 27] Шифр оперирует 128-битными блоками данных (пространство V_{128}) и использует ключ длиной $k = 256$ бит.

Алгоритм построен по архитектуре SP-сети (Substitution-Permutation network) и реализует 10 итераций преобразования состояния.

Структура алгоритма

Процесс зашифрования представляет собой композицию раундовых функций. Для $i = 1, \ldots, 9$ раунд состоит из трех последовательных преобразований LSX:

1. **Преобразование X (XOR)**: Побитовое сложение текущего состояния блока a с итерационным ключом K_i:

$$X[K_i](a) = K_i \oplus a$$

2. **Преобразование S (SubBytes)**: Нелинейное биективное преобразование. Вектор a разбивается на 16 байт, каждый из которых заменяется в соответствии с фиксированной перестановкой π (S-Box), заданной стандартом.

3. **Преобразование L (Linear transformation)**: Линейное перемешивание. Стандарт определяет его как композицию 16-кратного применения преобразования сдвига регистров R:

$$L(a) = R^{16}(a)$$

На практике (и в матричном представлении) это эквивалентно умножению вектора состояния на фиксированную матрицу M размером 16×16 над полем Галуа $GF(2^8)$ с неприводимым полиномом $p(x) = x^8 + x^7 + x^6 + x + 1$.

Последний (десятый) раунд является неполным и состоит только из наложения последнего итерационного ключа: $E(a) = X[K_{10}] \circ LSX[K_9] \circ \cdots \circ LSX[K_1](a)$.

Итерационные ключи $K_1, \ldots, K_{10}$ вырабатываются из мастер-ключа (256 бит) с помощью алгоритма развертывания, использующего сеть Фейстеля.

2.4.3. Режим MGM (Multilinear Galois Mode)

Так как «Кузнечик» является блочным шифром, а протокол `WireGuard` требует потокового шифрования с аутентификацией (AEAD), используется режим MGM (ГОСТ Р 1323565.1.026—2019). [23] Данный режим стандартизирован для использования с алгоритмом «Кузнечик» и работает по схеме «Encrypt-then-MAC».

Конфиденциальность

Шифрование осуществляется методом гаммирования (аналог режима CTR), что позволяет обрабатывать потоки данных произвольной длины без необходимости выравнивания (padding).

1. **Инициализация:** Формируется начальное значение счетчика T_1 на основе уникального вектора (Nonce) и ключа.

2. **Генерация гаммы:** Для каждого блока данных i вырабатывается блок гаммы Z_i путем зашифрования текущего значения счетчика:

$$Z_i = E_K(T_i), \quad T_{i+1} = \text{incr}(T_i)$$

   где incr — операция инкремента в кольце вычетов по модулю 2^{128}.

3. **Шифрование:** Шифротекст C_i получается сложением по модулю 2

блока открытого текста P_i с гаммой:

$$C_i = P_i \oplus \text{MSB}_{|P_i|}(Z_i)$$

Имитозащита

Для обеспечения целостности и аутентификации вычисляется имитовставка TAG с использованием мультилинейной функции. Особенностью MGM является использование схемы Вегмана-Картера, где для умножения каждого блока данных используется свой уникальный секретный множитель H_j.

Множители H_j вырабатываются путем шифрования значений счетчика, отличных от тех, что использовались для гаммирования (обеспечивается разделением домена, например, установкой старшего бита):

$$H_j = E_K(\text{Counter}_j^{\text{auth}})$$

Финальная имитовставка вычисляется как:

$$\text{TAG} = \text{MSB}_{\text{len}}\left(E_K(\text{Nonce}) \oplus \sum_j (A_j \otimes H_j) \oplus \sum_k (C_k \otimes H_{k+\ldots})\right)$$

где операции сложения и умножения ($\otimes$) выполняются в поле $GF(2^{128})$. Использование уникальных множителей H_j обеспечивает доказуемую стойкость режима и возможность параллельного вычисления имитовставки.

2.4.4. Протокол выработки общего ключа VKO ГОСТ Р 34.10-2012

Для замены протокола `Curve25519` используется алгоритм выработки общего ключа VKO (Выработка Ключа Обмена), реализуемый на эллиптических кривых, определенных в ГОСТ Р 34.10-2012. [24]

Математическая модель

Алгоритм оперирует элементами группы точек эллиптической кривой E,

заданной уравнением над конечным простым полем F_p:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

Параметры кривой (a, b, p, q, P, m) должны удовлетворять требованиям раздела 5.2 стандарта.

Пусть Участник А обладает парой ключей (d_A, Q_A), где:

- d_A — секретный ключ (целое число, $0 < d_A < q$);
- $Q_A = d_A P$ — открытый ключ (точка кривой), вычисленный как кратная точка от базовой точки P.

Аналогично, Участник В обладает парой (d_B, Q_B).

Процедура выработки общего секрета на стороне Участника А выглядит следующим образом:

1. **Валидация ключа партнера:** Проверяется, что точка Q_B удовлетворяет уравнению кривой и не является нулевой точкой O.
2. **Вычисление общей точки:** Вычисляется точка C (Common Point) умножением открытого ключа партнера на свой секретный ключ и кофактор кривой:
   $$C = (n \cdot d_A) \cdot Q_B$$
   где $n = m/q$ — кофактор (индекс подгруппы), обеспечивающий нахождение точки в подгруппе порядка q. В силу свойств группы точек эллиптической кривой участники получат одну и ту же точку:
   $$C = n \cdot d_A \cdot (d_B P) = n \cdot d_B \cdot (d_A P) = (n \cdot d_B) \cdot Q_A$$
3. **Вывод ключа (KDF):** Результирующий общий ключ K формируется путем хэширования x-координаты полученной точки x_C совместно с дополнительной ключевой информацией:
   $$K = h(UKM || \text{Vec}(x_C))$$

где:

- $h()$ — хэш-функция ГОСТ Р 34.11-2012 («Стрибог»);
- Vec(x_C) — двоичное представление координаты x точки C;
- UKM (User Keying Material) — 64-битное случайное число (Nonce), передаваемое открыто для обеспечения уникальности сессионного ключа.

3 Программная реализация и тестирование

Разработанный автором работы программный комплекс, реализующий виртуальный криптографический сетевой интерфейс, получил название `WireGost`. Ключевым архитектурным принципом разработки стала максимальная интеграция с существующей инфраструктурой ядра `Linux` и использование стандартных механизмов операционной системы.

С целью упрощения разработки, для всех используемых криптографических модулей ядра были написаны «функции-обёртки», позволяющие намного проще вносить изменения в реализацию модулей, без необходимости внесения изменений в основную часть кода модуля `WireGost`. (см. приложение **??**)

3.1 Реализация криптографических примитивов ГОСТ

3.1.1. Алгоритм хэширования ГОСТ Р 34.11-2012

Анализ исходного кода ядра `Linux` (для версии 5.10 и выше) показал, что алгоритм ГОСТ Р 34.11-2012 («Стрибог») не требует разработки, так как он уже является частью `Crypto API`. Несмотря на то, что изменение модуля не требовалось, автором было решено всё равно использовать промежуточный слой абстракции, упомянутый ранее, с целью того, чтобы держать кодовую базу в более однородном формате. Этот слой абстракции был расположен в файле `gost/gost_streebog.c`.

Инициализация и управление контекстом

Благодаря использованию промежуточного слоя абстракции, удалось незначительно оптимизировать работу модуля путем инициализации дескриптора алгоритма (`tfm` — transformation object) единожды при загрузке модуля, а не при каждом вычислении хэша.

Функция инициализации `gost_streebog_init_module` выполняет следующие критические операции:

1. запрос алгоритма `"streebog256"` через вызов `crypto_alloc_shash`;
2. контроль корректности выделения ресурсов;
3. проверка размера контекста хэширования. Поскольку структура состояния `struct gost_streebog_state` размещается на стеке ядра, необходимо гарантировать, что её размер не превышает зарезервированный буфер `GOST_STREEBOG_CTX_SIZE`, чтобы избежать переполнения стека.

Программный интерфейс

Реализован набор «функций-оберток», инкапсулирующих вызовы Crypto API:

- `gost_streebog256_init`: связывание контекста с глобальным дескриптором и сброс состояния;
- `gost_streebog256_update`: подача данных на вход функции сжатия;
- `gost_streebog256_final`: финализация вычислений и формирование дайджеста.

```
#include "gost_streebog.h"
#include <linux/err.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/string.h>

static struct crypto_shash *streebog_tfm;
```

```
int gost_streebog_init_module(void)
{
    streebog_tfm = crypto_alloc_shash("streebog256", 0, 0);
    if (IS_ERR(streebog_tfm)) {
        pr_err("WireGost: failed to allocate streebog256: %ld\n", PTR_ERR(streebog_t
        return PTR_ERR(streebog_tfm);
    }
    if (crypto_shash_descsize(streebog_tfm) > GOST_STREEBOG_CTX_SIZE) {
        crypto_free_shash(streebog_tfm);
        return -ENOMEM;
    }
    return 0;
}

void gost_streebog_cleanup_module(void)
{
    if (streebog_tfm) {
        crypto_free_shash(streebog_tfm);
        streebog_tfm = NULL;
    }
}

void gost_streebog256_init(struct gost_streebog_state *ctx)
{
    if (likely(streebog_tfm)) {
        ctx->desc.tfm = streebog_tfm;
        crypto_shash_init(&ctx->desc);
    }
}

void gost_streebog256_update(struct gost_streebog_state *ctx, const u8 *data, size_t
{
    crypto_shash_update(&ctx->desc, data, len);
}

void gost_streebog256_final(struct gost_streebog_state *ctx, u8 *hash)
{
    crypto_shash_final(&ctx->desc, hash);
}

void gost_streebog256(u8 *out, const u8 *in, size_t len)
{
```

```
    struct gost_streebog_state ctx;
    gost_streebog256_init(&ctx);
    gost_streebog256_update(&ctx, in, len);
    gost_streebog256_final(&ctx, out);
    memzero_explicit(&ctx, sizeof(ctx));
}
```

3.1.2. Конструкция HMAC на базе ГОСТ Р 34.11-2012

Оригинальный протокол `WireGuard` использует хэш-функцию `BLAKE2s`, поддерживающую режим «хэширования с ключом». Алгоритм «Стрибог», реализованный в ядре, является «чистой» хэш-функцией и не поддерживает инициализацию ключом.

Для обеспечения аутентификации сообщений и реализации функций формирования ключа (`KDF`) была программно реализована схема `HMAC` (`Hash-based Message Authentication Code`) в соответствии с `RFC 2104` [8] и рекомендациями `RFC 7836` [4].

HMAC-Стрибог был реализован в файле `gost/gost_hmac.c`, сам алгоритм при этом базируется на формуле:

$$\text{HMAC}(K, m) = H((K' \oplus opad) \parallel H((K' \oplus ipad) \parallel m))$$

Где K' — нормализованный ключ, $ipad$ и $opad$ — константы внутреннего и внешнего дополнения соответственно.

Особенности реализации

Ключевым требованием к реализации в пространстве ядра является безопасная работа с памятью. Функция `gost_hmac256` обеспечивает принудительную очистку всех буферов, содержащих производные ключа и промежуточные хэши, с помощью вызова `memzero_explicit`, исключая утечку ключевого материала.

```
void gost_hmac256(u8 *out, const u8 *in, size_t inlen,
                  const u8 *key, size_t keylen)
```

```
{
    struct gost_streebog_state state;
    u8 x_key[STREEBOG_BLOCK_SIZE];
    u8 i_hash[STREEBOG256_DIGEST_SIZE];
    u8 pad_key[STREEBOG_BLOCK_SIZE];
    int i;

    /* 1. Нормализация ключа до 64 байт */
    if (keylen > STREEBOG_BLOCK_SIZE) {
        gost_streebog256(x_key, key, keylen);
        memset(x_key + STREEBOG256_DIGEST_SIZE, 0,
               STREEBOG_BLOCK_SIZE - STREEBOG256_DIGEST_SIZE);
    } else {
        memcpy(x_key, key, keylen);
        if (keylen < STREEBOG_BLOCK_SIZE)
            memset(x_key + keylen, 0, STREEBOG_BLOCK_SIZE - keylen);
    }

    /* 2. inner_hash: H((K^ipad) || msg) */
    for (i = 0; i < STREEBOG_BLOCK_SIZE; ++i)
        pad_key[i] = x_key[i] ^ 0x36;

    gost_streebog256_init(&state);
    gost_streebog256_update(&state, pad_key, STREEBOG_BLOCK_SIZE);
    gost_streebog256_update(&state, in, inlen);
    gost_streebog256_final(&state, i_hash);

    /* 3. outer_hash: H((K^opad) || inner_hash) */
    for (i = 0; i < STREEBOG_BLOCK_SIZE; ++i)
        pad_key[i] = x_key[i] ^ 0x5c;

    gost_streebog256_init(&state);
    gost_streebog256_update(&state, pad_key, STREEBOG_BLOCK_SIZE);
    gost_streebog256_update(&state, i_hash, STREEBOG256_DIGEST_SIZE);
    gost_streebog256_final(&state, out);

    /* Очистка ключевой информации */
    memzero_explicit(x_key, sizeof(x_key));
    memzero_explicit(pad_key, sizeof(pad_key));
    memzero_explicit(i_hash, sizeof(i_hash));
}
```

3.1.3. Блочный шифр «Кузнечик» и режим MGM

В отличие от алгоритма хэширования реализация блочного шифра ГОСТ Р 34.12-2015 («Кузнечик») отсутствует в ядре Linux. Так что было принято решение разработать свой собственный загружаемый модуль ядра, который его реализует.
(см. приложение **??**)

Для обеспечения конфиденциальности и целостности данных, согласно требованиям протокола, используется режим аутентифицированного шифрования (AEAD). В соответствии с ГОСТ Р 34.13-2015 [26] был выбран режим гаммирования с мультилинейным преобразованием — `MGM` (Multilinear Galois Mode).

Оптимизация производительности

«Кузнечик» является шифром на основе `SP`-сети. Прямая программная реализация линейного преобразования L (умножение вектора на матрицу в поле $GF(2^8)$) обладает низкой производительностью. Для обеспечения пропускной способности, сравнимой с оригинальной версией `WireGuard`, была применена техника предварительно вычисленных таблиц.

Преобразование $L \circ S$ над 128-битным блоком заменяется на сумму результатов выборки из таблиц:

$$L(S(a_0, \ldots, a_{15})) = \bigoplus_{i=0}^{15} T_i[a_i]$$

где T_i — таблицы размером 256×128 бит.

При написании модуля были применены следующие техники оптимизации:

- **Предварительный расчет таблиц:** использование 16 таблиц для шифрования и 16 для расшифрования;
- **Развертка циклов:** полная развертка циклов внутри раундов шифрования;

- **Невыравненный доступ к буферам:** использование макросов `get_unaligned` / `put_unaligned` для работы с сетевыми буферами без предварительного копирования.

Реализация режима MGM

Режим `MGM` работает по схеме `Encrypt-then-MAC`. Ключевой операцией, влияющей на скорость, является умножение в поле Галуа $GF(2^{128})$ при вычислении имитовставки. В реализации задействована оптимизированная функция ядра `gf128mul_x_bbe` и 64-битные инструкции XOR.

```
/* Макрос для оптимизированного преобразования LSX с использованием T-таблиц.
 * Выполняет XOR результатов выборки из 16 таблиц (по одной на каждый байт).
 */
static __always_inline void LSX(u8 *a, const u8 *b, const u8 *c)
{
    u64 r0, r1;
    const u64 *t;

    /* Инициализация первой итерации (байт 0) */
    t  = (const u64 *)&kuz_table[0][(b[0] ^ c[0]) * 16];
    r0 = t[0];
    r1 = t[1];

    /* Развернутый цикл XOR для оставшихся 15 байт */
#define LSX_XOR(i) \
    t = (const u64 *)&kuz_table[i][(b[i] ^ c[i]) * 16]; \
    r0 ^= t[0]; \
    r1 ^= t[1];

    LSX_XOR(1);  LSX_XOR(2);  LSX_XOR(3);  LSX_XOR(4);
    LSX_XOR(5);  LSX_XOR(6);  LSX_XOR(7);  LSX_XOR(8);
    LSX_XOR(9);  LSX_XOR(10); LSX_XOR(11); LSX_XOR(12);
    LSX_XOR(13); LSX_XOR(14); LSX_XOR(15);

    /* Сохранение результата в выходной буфер как u64 (оптимизация) */
    ((u64 *)a)[0] = r0;
    ((u64 *)a)[1] = r1;
#undef LSX_XOR
}
```

```c
/* Функция шифрования одного блока (ECB).
 * Использует предварительно развернутые ключи и макрос LSX.
 */
static void kuznyechik_encrypt(struct crypto_tfm *tfm, u8 *out, const u8 *in)
{
    const struct crypto_kuznyechik_ctx *ctx = crypto_tfm_ctx(tfm);
    const u64 *keys = (const u64 *)ctx->key;
    u8 temp[KUZNYECHIK_BLOCK_SIZE] __aligned(sizeof(u64));
    u64 *o = (u64 *)out;
    u64 *t = (u64 *)temp;

    /* 9 раундов преобразований LSX */
    LSX(temp, (u8 *)(keys + 0), in);    /* Раунд 1 */
    LSX(temp, (u8 *)(keys + 2), temp);  /* Раунд 2 */
    LSX(temp, (u8 *)(keys + 4), temp);  /* Раунд 3 */
    /* ... */
    LSX(temp, (u8 *)(keys + 16), temp); /* Раунд 9 */

    /* Последний раунд: только сложение с ключом (X-преобразование) */
    o[0] = t[0] ^ keys[18];
    o[1] = t[1] ^ keys[19];
}

/* Быстрое обновление тега аутентификации MGM.
 * Использует встроенную функцию ядра для умножения в поле Галуа.
 */
static inline void mgm_mul_add_tag_fast(be128 *tag, const u8 *data,
                                        const be128 *h_i)
{
    be128 d_block;
    /* Оптимизация: чтение 64-битными словами, обработка невыровненных данных */
    const u64 *dp = (const u64 *)data;
    u64 *tp = (u64 *)&d_block;

    tp[0] = get_unaligned(&dp[0]);
    tp[1] = get_unaligned(&dp[1]);

    /* Аппаратно-оптимизированное умножение: d_block = d_block * h_i in GF(2^128) */
    gf128mul_x_bbe(&d_block, h_i);

    /* Аккумуляция результата в тег (XOR) */
```

```
    kuz_be128_xor(tag, tag, &d_block);
}
```

Слой абстракции AEAD

По аналогии с другими алгоритмами был также написан слой абстракции AEAD для шифрования и аутентификации данных. (gost/gost_kuznyechik.c)

```
/* Вспомогательная функция для выполнения операций AEAD.
 * Инкапсулирует работу со структурами ядра scatterlist и aead_request.
 */
static int do_aead_op(struct crypto_aead *tfm, u8 *dst, const u8 *src, size_t len,
                      const u8 *ad, size_t ad_len, const u8 *nonce, bool encrypt)
{
    struct aead_request *req;
    struct scatterlist sg_src[2], sg_dst[2];
    int ret;

    /*
     * Так как WireGuard обрабатывает пакеты в контексте SoftIRQ.
     * Использование GFP_KERNEL здесь привело бы к "sleep inside atomic section".
     * Поэтому обязательно используем GFP_ATOMIC.
     */
    req = aead_request_alloc(tfm, GFP_ATOMIC);
    if (!req)
        return -ENOMEM;

    /*
     * Формирование цепочек scatterlist (SG).
     * API ядра требует, чтобы ассоциированные данные (AD) шли перед
     * шифруемыми данными в виртуальном представлении памяти.
     */
    if (ad_len > 0) {
        /* Формируем входной SG: [AD] -> [SRC] */
        sg_init_table(sg_src, 2);
        sg_set_buf(&sg_src[0], ad, ad_len);
        sg_set_buf(&sg_src[1], src, len);

        /* Формируем выходной SG: [AD] -> [DST] */
        sg_init_table(sg_dst, 2);
```

```
        sg_set_buf(&sg_dst[0], ad, ad_len);
        /* При шифровании выходной буфер должен вмещать данные + тег (16 байт) */
        sg_set_buf(&sg_dst[1], dst, len + (encrypt ? KUZNYECHIK_MGM_TAG_SIZE : 0));

        aead_request_set_crypt(req, sg_src, sg_dst, len, (u8 *)nonce);
        aead_request_set_ad(req, ad_len);
    } else {
        /* Упрощенный случай без AD (линейное отображение) */
        sg_init_one(sg_src, src, len);
        sg_init_one(sg_dst, dst, len + (encrypt ? KUZNYECHIK_MGM_TAG_SIZE : 0));

        aead_request_set_crypt(req, sg_src, sg_dst, len, (u8 *)nonce);
        aead_request_set_ad(req, 0);
    }

    /* Выполняем операцию. В данном случае вызов синхронный,
     * так как используется программная реализация шифра.
     */
    if (encrypt)
        ret = crypto_aead_encrypt(req);
    else
        ret = crypto_aead_decrypt(req);

    aead_request_free(req);
    return ret;
}

/* Публичный интерфейс для шифрования, используемый в send.c */
int gost_kuznyechik_mgm_encrypt(struct gost_kuznyechik_mgm_ctx *ctx,
                                u8 *dst, const u8 *src, size_t src_len,
                                const u8 *ad, size_t ad_len, const u8 *nonce)
{
    if (unlikely(!ctx->tfm))
        return -EINVAL;
    return do_aead_op(ctx->tfm, dst, src, src_len, ad, ad_len, nonce, true);
}
```

3.1.4. Протокол выработки общего ключа VKO ГОСТ Р 34.10-2012

В качестве замены алгоритму ECDH (Curve25519) реализован протокол VKO (Выработка Ключа Обмена) на базе эллиптических кривых. Модуль ec256_vko регистрируется в ядре как примитив типа KPP (Key-agreement

Protocol Primitive). (см. приложение **??**)

Параметры алгоритма

Используется 256-битная эллиптическая кривая с параметрами `id-tc26-gost-3410-2012-256-paramSetA`. Вычисление общей точки производится по формуле:

$$K = (h \cdot d_{local}) \times Q_{remote}$$

Для арифметических операций с большими числами (`BigInt`) адаптирована библиотека `crypto/ecc.c`, обеспечивающая выполнение операций за постоянное время, что защищает от атак по побочным каналам.

Программная реализация

Особенностью программной реализации является работа с 64-байтными открытыми ключами (две координаты по 32 байта) и поддержка параметра `UKM` (`User Keying Material`). Функция `gost_vko_compute_value` выполняет декодирование ключей, скалярное умножение и хэширование результата для получения сессионного ключа.

```
/* Основная функция вычисления общего секрета (VKO).
 * Выполняет роль callback-функции для KPP API ядра.
 */
static int gost_vko_compute_value(struct kpp_request *req)
{
    struct crypto_kpp *tfm = crypto_kpp_reqtfm(req);
    struct gost_vko_ctx *ctx = gost_vko_get_ctx(tfm);
    u64 scalar[GOST_EC256_NDIGITS];

    /* ... */

    if (req->src) {
        /* --- Режим вычисления общего секрета (Shared Secret) --- */

        /* 1. Декодирование публичного ключа партнера (точка Q).
         * Ключ приходит в формате Big-Endian (64 байта), преобразуем
         * во внутренний формат VLI (массив u64).
```

```
         */
        /* Координата X */
        gost_vli_from_be(public_key_buf, req->src, GOST_EC256_NDIGITS);
        /* Координата Y (смещение 32 байта) */
        gost_vli_from_be(public_key_buf + GOST_EC256_NDIGITS,
                         req->src + GOST_EC256_KEY_SIZE, GOST_EC256_NDIGITS);

        /* 2. Вычисление эффективного скаляра S.
         * Согласно ГОСТ Р 34.10-2012:
         * Если передан UKM (User Keying Material), то S = (d * UKM) mod n.
         * Иначе S = d (собственный закрытый ключ).
         */
        if (ctx->params.ukm_size > 0) {
            gost_vli_mod_mult(scalar, ctx->params.key, ctx->params.ukm,
                              ctx->curve->n, GOST_EC256_NDIGITS);
        } else {
            memcpy(scalar, ctx->params.key, sizeof(scalar));
        }

        /* 3. Скалярное умножение точки кривой.
         * Вычисляем точку K = S * Q_peer.
         * Функция использует алгоритм с защитой от атак по времени.
         */
        gost_ec256_point_mul(shared_secret_x,     /* Результат (Координата X) */
                             temp_y,              /* Результат Y (отбрасывается) */
                             public_key_buf,      /* Peer X */
                             public_key_buf + GOST_EC256_NDIGITS, /* Peer Y */
                             scalar, ctx->curve);

        /* 4. Формирование KEK (Key Encryption Key).
         * Согласно алгоритму VKO, общим секретом является хэш от
         * координаты X полученной точки: KEK = Hash(K.x)
         */
        /* ... Вызов хэширования streebog256(shared_secret_x) ... */
    }
    else {
        /* --- Режим генерации собственного публичного ключа --- */
        /* Q = d * G, где G - базовая точка кривой */
        gost_ec256_point_mul(public_key_buf,
                             public_key_buf + GOST_EC256_NDIGITS,
                             ctx->curve->g.x, ctx->curve->g.y,
                             ctx->params.key, ctx->curve);
```

```
    }

    /* ... Копирование результата в req->dst ... */
    return 0;
}
```

3.2 Интеграция в сетевой протокол WireGuard

Дальнейшим шагом разработки `WireGost` стало внедрение ранее реализованных криптографических примитивов в сетевой протокол `WireGuard`. Этот шаг оказался довольно сложным, поскольку пришлось переписывать значительную часть кода, связанного с обработкой пакетов, а также общением с пользовательским пространством через `Netlink`.

3.2.1. Адаптация структур данных

Первым этапом портирования стало изменение размера криптографических ключей, так как реализованный алгоритм `VKO` использует ключи размером 64 байта, что в 2 раза больше, чем в оригинальном протоколе:

- **Публичные ключи:** увеличены с 32 до 64 байт (нескомпрессированная точка);
- **Размер пакета:** изменения отражены в структурах `message_handshake_initiation` и `message_handshake_response` в файле `messages.h`.

Увеличение заголовков пакетов рукопожатия было учтено при расчете `MTU` туннеля, чтобы избежать фрагментации пакетов на этапе установления соединения.

3.2.2. Модификация машины состояний Noise

Ядро протокола (`noise.c`) было адаптировано для использования API разработанных модулей. Функция `mix_dh`, отвечающая за обновление цепочки ключей (chaining key), переписана для вызова `gost_ec256_dh`. Вместо прямой математической операции теперь происходит обращение к KPP API ядра:

```
/* Адаптированная функция mix_dh для VKO ГОСТ */
static bool mix_dh(u8 *chaining_key, u8 *key, struct wg_peer *peer)
{
    u8 shared_secret[NOISE_HASH_LEN];
    /* Вызов KPP API для вычисления VKO */
    if (!gost_ec256_dh(shared_secret,
                       peer->handshake.static_private,
                       peer->handshake.remote_static,
                       peer->handshake.ukm,
                       peer->handshake.ukm_len))
        return false;

    /* KDF на базе HMAC-Streebog */
    kdf(chaining_key, key, NULL, shared_secret, ...);
    memzero_explicit(shared_secret, NOISE_HASH_LEN);
    return true;
}
```

3.2.3. Взаимодействие через Netlink

Так как был изменен размер ключа, в протоколе `Netlink` были обновлены атрибуты (`WGDEVICE_A_PEER_PUBLIC_KEY`, `WGDEVICE_A_PRIVATE_KEY`), а в драйвер добавлены проверки, не позволяющие конфигурировать интерфейс ключами некорректной длины.

3.2.4. Инициализация и управление зависимостями

Из-за того, что в модуль `WireGost` был добавлен слой абстракции, для каждого из используемых криптографических алгоритмов необходимо было выполнить инициализацию и деинициализацию. Для этого в `main.c` были добавлены вызовы `gost_streebog_init_module`, `gost_kuznyechik_init_tfms`, `gost_ec256_init_module`.

Более того, был расширен набор тестов, которые выполняются при сборке в режиме `DEBUG`. Были добавлены встроенные тесты, проверяющие корректность работы алгоритмов на эталонных векторах:

```
#ifdef DEBUG
    RUN_TEST("Streebog-256", self_test_streebog);
    RUN_TEST("HMAC-Streebog-256", self_test_hmac_streebog);
```

```
    RUN_TEST("Kuznyechik-MGM", self_test_kuznyechik_mgm);
    RUN_TEST("GOST R 34.10-2012 (EC512)", self_test_gost_vko_generated);

    pr_info("WireGost crypto self-tests completed successfully.\n");
#endif
```

Сами тесты были вынесены в `selftest/selftest.c` к остальным файлам самотестирования.

Для упрощения добавления новых тестов был создан макрос `RUN_TEST`, который автоматически генерирует код для запуска теста.

```
#define RUN_TEST(name, func)            \
do                                      \
{                                       \
    pr_info("Testing " name "... ");    \
    if (func())                         \
        pr_cont("PASSED\n");            \
    else                                \
    {                                   \
        pr_cont("FAILED\n");            \
        ret = -EFAULT;                  \
        goto err_tests;                 \
    }                                   \
} while (0)
```

Сами тесты представляют собой набор эталонных значений, определенных в ГОСТ соответствующего протокола. [25] Ниже представлен пример теста для алгоритма `Streebog-256`.

```
static bool self_test_streebog(void)
{
    u8          digest[32];
    const char *msg          = "012345678901234567890123456789012"
                               "345678901234567890123456789012";
    const u8    expected[32] = {0x9d, 0x15, 0x1e, 0xef, 0xd8, 0x59,
                                0x0b, 0x89, 0xda, 0xa6, 0xba, 0x6c,
                                0xb7, 0x4a, 0xf9, 0x27, 0x5d, 0xd0,
                                0x51, 0x02, 0x6b, 0xb1, 0x49, 0xa4,
                                0x52, 0xfd, 0x84, 0xe5, 0xe5, 0x7b,
                                0x55, 0x00};
```

```
    gost_streebog256(digest, (u8 *)msg, 63);

    if (memcmp(digest, expected, 32) != 0)
    {
        pr_err("Streebog-256 digest mismatch\n");
        return false;
    }
    return true;
}
```

3.2.5. Конфигурация протокола Noise_GOST

В файле `noise.c` определена уникальная строка идентификации протокола:

```
"Noise_IKpsk2_GOST_EC512_KuznyechikMGM_Streebog256"
```

Это гарантирует криптографическую изоляцию сессий `WireGost` от оригинального `WireGuard`, предотвращая попытки соединения с несовместимыми клиентами. Функции деривации ключей (`KDF`) переведены на использование конструкции HMAC-Streebog256.

Функция формирования ключа (KDF)

Стандартная реализация `KDF` в протоколе Noise базируется на `HKDF` (HMAC-based Key Derivation Function). В разработанном модуле функция была адаптирована для использования `HMAC` на базе ГОСТ Р 34.11-2012:

```
static void kdf(u8 *first_dst, u8 *second_dst, u8 *third_dst,
                const u8 *data, size_t first_len, size_t second_len,
                size_t third_len, size_t data_len,
                const u8 chaining_key[NOISE_HASH_LEN])
{
    u8 output[NOISE_HASH_LEN + 1];
    u8 secret[NOISE_HASH_LEN];

    /* Шаг 1: Извлечение */
    gost_hmac256(secret, data, data_len, chaining_key, NOISE_HASH_LEN);
```

```
    /* Шаг 2: Расширение для первого ключа */
    if (first_dst && first_len) {
        output[0] = 1;
        gost_hmac256(output, output, 1, secret, NOISE_HASH_LEN);
        memcpy(first_dst, output, first_len);
    }

    /* Расширение для второго ключа */
    if (second_dst && second_len) {
        output[NOISE_HASH_LEN] = 2;
        gost_hmac256(output, output, NOISE_HASH_LEN + 1, secret, NOISE_HASH_LEN);
        memcpy(second_dst, output, second_len);
    }

    /* ... (аналогично для третьего ключа) ... */

    /* Очистка ключевого материала из памяти */
    memzero_explicit(secret, NOISE_HASH_LEN);
    memzero_explicit(output, NOISE_HASH_LEN + 1);
}
```

Обертка шифрования сообщений

Для интеграции блочного шифра «Кузнечик» в режиме MGM в поток обработки пакетов были реализованы функции-обертки `message_encrypt` и `message_decrypt`. Они выполняют следующие задачи:

1. инициализация контекста шифрования сессионным ключом;
2. вызов процедуры AEAD-шифрования;
3. обновление текущего хэша рукопожатия шифртекстом, что обеспечивает криптографическое связывание всех сообщений сессии.

```
static void message_encrypt(u8 *dst_ciphertext, const u8 src_plaintext,
                            size_t src_len, u8 key[NOISE_SYMMETRIC_KEY_LEN],
                            u8 hash[NOISE_HASH_LEN])
{
    u8 nonce[16] = {0};
    struct gost_kuznyechik_mgm_ctx ctx;
    memset(&ctx, 0, sizeof(ctx));
```

```
    /* Установка ключа и шифрование */
    if (likely(gost_kuznyechik_mgm_set_key(&ctx, key) == 0))
    {
        gost_kuznyechik_mgm_encrypt(&ctx, dst_ciphertext, src_plaintext,
                                    src_len, hash, NOISE_HASH_LEN, nonce);
        gost_kuznyechik_mgm_free_ctx(&ctx);
    }

    /* MixHash: H = Hash(H || Ciphertext) */
    mix_hash(hash, dst_ciphertext, noise_encrypted_len(src_len));
}

static bool message_decrypt(u8 *dst_plaintext, const u8 *src_ciphertext,
                            size_t src_len, u8 key[NOISE_SYMMETRIC_KEY_LEN],
                            u8 hash[NOISE_HASH_LEN])
{
    u8                              nonce[16] = {0};
    bool                           ret         = false;
    struct gost_kuznyechik_mgm_ctx ctx;
    memset(&ctx, 0, sizeof(ctx));
    if (likely(gost_kuznyechik_mgm_set_key(&ctx, key) == 0))
    {
        if (gost_kuznyechik_mgm_decrypt(&ctx, dst_plaintext, src_ciphertext,
                                        src_len, hash, NOISE_HASH_LEN, nonce)
            == 0)
        {
            ret = true;
        }
        gost_kuznyechik_mgm_free_ctx(&ctx);
    }
    if (!ret)
        return false;
    mix_hash(hash, src_ciphertext, src_len);
    return true;
}
```

Инкапсуляция и шифрование

Процесс отправки пакета реализован в файле `send.c`.

Перед шифрованием данные должны быть подготовлены:

1. **Выравнивание:** для скрытия точной длины передаваемых данных и соблюдения требований блочного шифра, к пакету добавляется случайное заполнение. Размер выравнивания рассчитывается так, чтобы итоговая длина блока соответствовала требованиям `MTU` и границам блока шифра;

2. **Формирование заголовка:** создается структура `message_data`, содержащая тип сообщения (`MESSAGE_DATA`), индекс ключа получателя и 64-битный счетчик (Nonce), используемый для защиты от атак повтора.

Функция `encrypt_packet` выполняет эти действия и вызывает криптографический примитив:

```
static bool encrypt_packet(struct sk_buff *skb, struct noise_keypair *keypair)
{
    unsigned int padding_len, plaintext_len, trailer_len;
    struct message_data header;
    /* ... */
    u8 nonce[16] = {0};
    /* Расчет выравнивания и общей длины текста */
    padding_len   = calculate_skb_padding(skb);
    /* Длина трейлера включает имитовставку (Tag) */
    trailer_len   = padding_len + noise_encrypted_len(0);
    plaintext_len = skb->len + padding_len;

    /* ... подготовка буфера skb (skb_cow_data) ... */

    /* Формирование заголовка */
    header              = (struct message_data *)skb_push(skb, sizeof(*header));
    header->header.type = cpu_to_le32(MESSAGE_DATA);
    header->key_idx     = keypair->remote_index;
    header->counter     = cpu_to_le64(PACKET_CB(skb)->nonce);

    /* Использование 64-битного счетчика как части Nonce для MGM */
    memcpy(nonce, &header->counter, 8);

    /* ... линеаризация буфера ... */

    /* Шифрование "на месте" (in-place) с использованием Кузнечик-MGM */
```

```
    if (gost_kuznyechik_mgm_encrypt(&keypair->sending.mgm_ctx,
                                    skb->data + sizeof(struct message_data),
                                    skb->data + sizeof(struct message_data),
                                    plaintext_len, NULL, 0, nonce))
    {
        return false;
    }

    return true;
}
```

Инкапсуляция и шифрование

При получении пакета (файл `receive.c`) драйвер сначала определяет тип сообщения. Если это `MESSAGE_DATA`, управление передается функции `decrypt_packet`.

Процедура расшифрования включает:

1. **Проверку счетчика:** извлечение значения `counter` из заголовка и проверка, не является ли пакет дубликатом (`Replay Attack Protection`);
2. **Расшифрование с аутентификацией:** вызов функции `gost_kuznyechik_mgm_decrypt`. Режим `MGM` работает по схеме `AEAD`, поэтому проверка целостности (сверка имитовставки) происходит одновременно с расшифрованием. Если имитовставка не совпадает, пакет немедленно отбрасывается.

```
static bool decrypt_packet(struct sk_buff *skb, struct noise_keypair keypair)
{
    u8 nonce[16] = {0};
    /* ... */

    /* Валидация состояния ключа и счетчиков */
    if (unlikely(!READ_ONCE(keypair->receiving.is_valid) || ... )) {
        return false;
    }

    /* Извлечение Nonce из заголовка пакета */
    PACKET_CB(skb)->nonce = le64_to_cpu(((struct message_data *)skb->data)->counter)
```

```
    memcpy(nonce, &((struct message_data *)skb->data)->counter, 8);

    /* ... подготовка буфера ... */

    /* Дешифрование и проверка имитовставки */
    if (gost_kuznyechik_mgm_decrypt(&keypair->receiving.mgm_ctx,
                                    skb->data + sizeof(struct message_data),
                                    skb->data + sizeof(struct message_data),
                                    skb->len - sizeof(struct message_data),
                                    NULL, 0, nonce))
    {
        return false;
    }

    /* Удаление заголовка WireGuard и имитовставки (trim) */
    if (pskb_trim(skb, skb->len - NOISE_AUTHTAG_LEN))
        return false;
    skb_pull(skb, sizeof(struct message_data));

    return true;
}
```

3.3 Разработка инструментария конфигурации и тестирования

Поскольку внедрение алгоритмов описанных в ГОСТ (в частности ГОСТ 34.10-2012 и ГОСТ 34.11-2012) [24; 25] потребовало изменения длины криптографических ключей (увеличение с 32 до 64 байт для закрытого ключа и до 128 байт для открытого), стандартные утилиты пространства пользователя, такие как `wg` и `wg-quick`, оказались несовместимы с разработанным модулем `WireGost`.

Для отладки и тестирования взаимодействия с модулем ядра была разработана специализированная утилита командной строки `wg_gost_cli`.

3.3.1. Архитектура утилиты конфигурации

Утилита реализована на языке C и взаимодействует с ядром через интерфейс `Generic Netlink`. В отличие от использования библиотеки `libmnl`,

которая применяется в оригинальном WireGuard, здесь реализовано прямое формирование сообщений Netlink для обеспечения полного контроля над структурой атрибутов.

```
/* testing/wg_gost_cli.c */
#define WG_GENL_NAME "wiregost"

/* Размер закрытого ключа VKO (512 бит) /
#define GOST_PRIV_KEY_LEN 64
/ Размер открытого ключа (точка на эллиптической кривой, 64+64 байт) */
#define GOST_PUB_KEY_LEN 128

/* ... определения атрибутов Netlink (WGDEVICE_A_, WGPEER_A_) ... */
```

3.3.2. Инициализация интерфейса и генерация ключей

Основной задачей утилиты является инициализация сетевого интерфейса и генерация ключевой пары. Утилита поддерживает команду init, которая выполняет следующие действия:

1. преобразование имени интерфейса в индекс (if_nametoindex);
2. генерация случайного закрытого ключа;
3. отправка команды WG_CMD_SET_DEVICE с атрибутом WGDEVICE_A_PRIVATE_KEY;
4. отправка запроса WG_CMD_GET_DEVICE для получения вычисленного ядром открытого ключа.

Реализация отправки конфигурации:

```
int cmd_init(int sock, int family_id, int argc, char *argv)
{
    /* ... инициализация переменных ... */

    /* Генерация тестового ключа (обнуление старшего байта для валидности) */
    gen_random_key(priv, GOST_PRIV_KEY_LEN, getpid() + port);

    /* Формирование запроса SET_DEVICE */
```

```
    req.n.nlmsg_len   = NLMSG_LENGTH(GENL_HDRLEN);
    req.n.nlmsg_type  = family_id;
    req.n.nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
    req.g.cmd         = WG_CMD_SET_DEVICE;

    /* Добавление атрибутов: индекс, приватный ключ, порт */
    add_attr(&req, WGDEVICE_A_IFINDEX, &ifindex, sizeof(ifindex));
    add_attr(&req, WGDEVICE_A_PRIVATE_KEY, priv, GOST_PRIV_KEY_LEN);
    add_attr(&req, WGDEVICE_A_LISTEN_PORT, &port, sizeof(port));

    /* Отправка в ядро */
    if (send(sock, &req, req.n.nlmsg_len, 0) < 0) { /* ... */ }

    /* ... чтение ответа и запрос публичного ключа ... */
}
```

3.3.3. Настройка пиров

Вторая ключевая функция — добавление доверенных узлов (пиров). Команда `peer` формирует сложное вложенное сообщение `Netlink`, содержащее структуру `WGDEVICE_A_PEERS`, внутри которой находится список пиров с их параметрами: открытый ключ (128 байт), конечная точка (Endpoint) и разрешенные IP-адреса.

Особенностью реализации является корректное формирование вложенных атрибутов (`NLA_F_NESTED`), что необходимо для корректного парсинга со стороны ядра.

```
int cmd_peer(int sock, int family_id, int argc, char *argv)
{
    /* ... парсинг аргументов командной строки ... */
    /* Начало вложенного атрибута списка пиров */
    struct nlattr *peers = (struct nlattr *)((char *)&req + req.n.nlmsg_len);
    peers->nla_type      = WGDEVICE_A_PEERS | NLA_F_NESTED;
    req.n.nlmsg_len     += NLA_HDRLEN;

    /* Начало атрибута конкретного пира */
    struct nlattr *peer0 = (struct nlattr *)((char *)&req + req.n.nlmsg_len);
    peer0->nla_type      = 0 | NLA_F_NESTED; /* Индекс 0 */
    req.n.nlmsg_len     += NLA_HDRLEN;
```

```
    /* Установка параметров пира */
    add_attr(&req, WGPEER_A_PUBLIC_KEY, pub, GOST_PUB_KEY_LEN);
    add_attr(&req, WGPEER_A_ENDPOINT, &ep4, sizeof(ep4));

    /* ... добавление разрешенных IP (AllowedIPs) ... */

    /* Финализация длин вложенных атрибутов */
    peer0->nla_len = (char *)&req + req.n.nlmsg_len - (char *)peer0;
    peers->nla_len = (char *)&req + req.n.nlmsg_len - (char *)peers;

    /* Отправка конфигурации в ядро */
    if (send(sock, &req, req.n.nlmsg_len, 0) < 0) { /* ... */ }

    return 0;
}
```

3.4 Методология тестирования

Для объективной оценки производительности разработанного решения `WireGost` и сравнения его с оригинальным протоколом `WireGuard`, с `go` версией `ru-WireGuard` на ГОСТ протоколах, а также с эталонной пропускной способностью канала, был разработан автоматизированный комплекс скриптов на языке `Bash`. (см. приложение **??**)

3.4.1. Тестовая среда и топология

Тестирование проводилось в изолированной среде с использованием механизма сетевых пространств имен Linux (`Network Namespaces`). Это позволяет эмулировать полноценное сетевое взаимодействие между двумя узлами на одной физической машине, исключая влияние внешнего сетевого оборудования, но сохраняя полный путь прохождения пакетов через сетевой стек ядра.

Топология тестового стенда формируется функцией `setup_namespaces` в библиотеке `lib_benchmark.sh`:

- создаются два пространства имен: `ns1` (клиент) и `ns2` (сервер);
- они соединяются парой виртуальных ethernet-интерфейсов (`veth`),

эмулирующих физический канал связи (Underlay network, подсеть 10.0.0.0/24);

- поверх физического канала поднимаются VPN-интерфейсы (Overlay network, подсеть 192.168.1.0/24).

```
setup_namespaces() {
    echo -e "${GREEN}[Setup] Creating Namespaces and VETH...${NC}"
    ip netns add ns1
    ip netns add ns2

    ip link add veth1 type veth peer name veth2
    ip link set veth1 netns ns1
    ip link set veth2 netns ns2

    # Настройка транспортной сети (Underlay)
    ip netns exec ns1 bash -c "ip addr add 10.0.0.1/24 dev veth1; \
        ip link set veth1 up"
    ip netns exec ns2 bash -c "ip addr add 10.0.0.2/24 dev veth2; \
        ip link set veth2 up"
}
```

3.4.2. Архитектура тестового комплекса

Комплекс построен по модульному принципу. Основная логика измерений вынесена в библиотеку `lib_benchmark.sh`, которая предоставляет унифицированный интерфейс для запуска тестов. Специфичные для реализации настройки (настройка WireGuard, ruWireGuard, WireGost или «чистого» канала) вынесены в отдельные сценарии, переопределяющие функцию `setup_vpn_impl`.

Сценарий тестирования включает следующие этапы:

1. **Тест пропускной способности TCP и эффективности CPU:**
   Для тестирования используется утилита `iperf3`. Кроме измерения скорости (Mbit/s), рассчитывается «стоимость» передачи данных для процессора.
   Метрика эффективности рассчитывается по формуле:

$$Efficiency = \frac{Throughput_{Mbps}}{CPU_{usage}}$$

Где CPU_{usage} — суммарное потребление ресурсов ядра и пользователя за время теста.

```
run_tcp_efficiency_test() {
    # ... запуск сервера и клиента iperf3 ...
    local cpu_diff=$((cpu_end - cpu_start))

    # Расчет удельной стоимости трафика
    CPU_COST=$(echo "scale=2; ($cpu_diff / $time_diff_sec)" | bc)
    EFFICIENCY=$(echo "scale=2; $TCP_MBPS / $CPU_COST" | bc)

    echo -e "    Throughput: ${CYAN}${TCP_MBPS} Mbps${NC}"
    echo -e "    Efficiency: ${CYAN}${EFFICIENCY}${NC}"
}
```

2. **Тест максимальной пропускной способности UDP:**
   В отличие от TCP, протокол UDP не имеет встроенных механизмов контроля перегрузки. Запуск `iperf3` в режиме UDP с флагом `-b 0` (максимальная полоса) позволяет оценить «чистую» производительность канала и процент потери пакетов (`Packet Loss`), возникающих из-за переполнения буферов или нехватки процессорного времени на шифрование.

```
run_udp_saturation_test() {
    echo -e "${GREEN}[Test] UDP Max Throughput...${NC}"
    ip netns exec ns1 iperf3 -c 192.168.1.2 -u -b 0 -t $TEST_DURATION \
        -J > "$LOG_DIR/iperf_udp.json"

    # ... парсинг JSON для получения пропускной способности и потерь ...

    echo -e "    Throughput: ${CYAN}${UDP_MBPS} Mbps${NC}"
    echo -e "    Packet Loss:${CYAN}${UDP_LOSS}%${NC}"
}
```

3. **Стресс-тестирование PPS (Packets Per Second):**
   Отправка потока мелких UDP-пакетов (64 байта) для оценки накладных расходов на обработку заголовков, прерываний и вызовов крипто-

графических функций на каждый пакет, что является узким местом для тяжеловесных алгоритмов шифрования.

```
run_udp_pps_test() {
    echo -e "${GREEN}[Test] UDP PPS (64 byte packets)...${NC}"
    # Отправка пакетов по 64 байта
    ip netns exec ns1 iperf3 -c 192.168.1.2 -u -l 64 -b 2G -t 5 \
        -J > "$LOG_DIR/iperf_pps.json"

    # ... расчет среднего PPS ...
    echo -e "    Result:     ${CYAN}${PPS_RATE} pps${NC}"
}
```

4. **Тест целостности данных:**
   Для проверки корректности реализации криптографических алгоритмов (особенно режимов сцепления блоков и обработки счетчиков) был разработан тест на целостность передаваемых данных. Для этого из `/dev/urandom` генерируется файл размером 512 Мб, происходит передача файла через туннель, после чего сверяются контрольные суммы `MD5` на стороне отправителя и получателя.

```
run_file_integrity_test() {
    echo -e "${GREEN}[Test] File Integrity (${FILE_SIZE_MB} MB)...${NC}"
    # Генерация случайного файла
    dd if=/dev/urandom of="$test_file" bs=1M count=$FILE_SIZE_MB \
        status=none
    local orig_sum=$(md5sum "$test_file" | awk '{print $1}')

    # Передача через сокет (socat/nc) внутри туннеля
    # ... передача ...

    # Сверка хеш-сумм
    if [ "$orig_sum" == "$recv_sum" ]; then
        echo -e "    Result:     ${CYAN}PASS${NC}"
    else
        echo -e "    Result:     ${RED}FAIL${NC} (Checksum Mismatch)"
    fi
}
```

5. **Тест задержки под нагрузкой (Latency):**
   Измерение времени отклика (`RTT`) с помощью утилиты `ping`, выпол-

няемое параллельно с фоновой генерацией трафика (UDP-поток). Это позволяет оценить влияние очередей и времени шифрования на интерактивность соединения (эффект Bufferbloat).

```
run_latency_test() {
    echo -e "${GREEN}[Test] Latency under Load...${NC}"
    # Фоновая нагрузка
    ip netns exec ns1 iperf3 -c 192.168.1.2 -u -b 100M -t 10 \
        >/dev/null 2>&1 &
    local load_pid=$!
    sleep 2
    # Измерение задержки
    ip netns exec ns1 ping -c 5 192.168.1.2 > "$LOG_DIR/latency.log"
    wait $load_pid || true
    # ... парсинг среднего RTT ...
    echo -e "    Result:     ${CYAN}${LATENCY_AVG} ms${NC}"
}
```

3.5 Результаты тестирования

Тестирование производительности проводилось на аппаратной платформе со следующими характеристиками:

- **Процессор:** Intel Core i7-1360P (13-е поколение, архитектура Raptor Lake-P), 16 потоков, максимальная тактовая частота 5.0 ГГц;
- **Оперативная память:** 16 ГБ LPDDR5;
- **ОС и Ядро:** Arch Linux, ядро Linux 6.18.1-zen.

Для минимизации влияния планировщика процессов и фоновых задач каждый тест запускался по 10 раз, в качестве результатов пошли усредненные показатели.

3.5.1. Эталонные показатели (Baseline)

Измерения «чистого» виртуального канала (`veth`) без шифрования показали теоретический максимум производительности сетевого стека в данном окружении.

- **TCP пропускная способность:** ≈ 80.8 Гбит/с;

- **UDP пропускная способность:** ≈ 9.6 Гбит/с;
- **Стоимость CPU:** 237.96 `Jiffies`.

Такие высокие показатели обусловлены тем, что для передачи данных для вычисления эталонных показателей не требуется никаких криптографических преобразований.

3.5.2. Сравнительный анализ реализаций WireGuard

В таблице 1 приведены сводные результаты тестирования оригинального протокола `WireGuard`, `ruWireGuard` и разработанной автором реализации `WireGost`. (см. приложение **??**)

Таблица 1: Сводная таблица результатов тестирования производительности

Метрика	**WireGuard**	**ruWireGuard**	**WireGost**	**Δ (WG)**	**Δ (ruWG)**
TCP T-put (Mbps)	5503.61	61.54	743.12	$\downarrow 7.4\times$	$\uparrow 12.1\times$
UDP T-put (Mbps)	3553.11	39.97	845.72	$\downarrow 4.2\times$	$\uparrow 21.2\times$
UDP Loss (%)	0.1714%	98.5195%	13.1979%	$\downarrow 77\times$	$\uparrow 7.5\times$
CPU Cost (units)	499.05	1414.62	664.09	$\downarrow 1.3\times$	$\uparrow 2.1\times$
Efficiency (Mbps/CPU)	11.03	0.04	1.11	$\downarrow 9.9\times$	$\uparrow 27.8\times$
Latency (ms)	0.098	441.522	0.083	$\uparrow 1.2\times$	$\uparrow 5300\times$

**Δ (WG) — отношение WireGost к WireGuard;*

***Δ (ruWG) — отношение WireGost к ruWireGuard.*

Важно отметить, что при тестировании `ruWireGuard` было проведено не 10, а 11 тестов, так как один из них привел к аварийному завершению. К сожалению, так как тестирование проводилось в автоматическом режиме, получить подробную информацию о причине аварийного завершения не удалось, как и повторить условия, при которых оно возникло.

На основе полученных данных можно сделать следующие выводы о характеристиках разработанного автором программного комплекса:

Сравнение `WireGost` с оригинальным `WireGuard`

Как и ожидалось, внедрение российских криптографических алгоритмов привело к снижению пропускной способности. `WireGost` показал результат в 743 Мбит/с (TCP), что в 7.4 раза медленнее оригинального протокола

(5.5 Гбит/с). Это объясняется фундаментальными различиями используемых примитивов:

- **Природа шифра:** `WireGuard` использует потоковый шифр `ChaCha20`, который выполняет минимальное количество операций над памятью (`XOR` с гаммой) и идеально оптимизирован для программной реализации. `WireGost` использует блочный шифр «Кузнечик» (ГОСТ Р 34.12-2015). Программная реализация блочного шифра требует выполнения ресурсоемких раундовых преобразований (S-box, линейное преобразование) для каждого блока данных (16 байт);
- **Отсутствие SIMD-оптимизаций:** Оригинальный модуль `WireGuard` активно использует векторные инструкции процессора (AVX2/AVX-512) для параллельной обработки блоков. Текущая реализация `WireGost` выполнена на чистом Си без использования ассемблерных вставок, что оставляет значительный потенциал для оптимизации в будущем.

Тем не менее, достигнутая скорость в ≈ 750 Мбит/с является достаточной для насыщения большинства каналов связи (100 Мбит/с и приближается к 1 Гбит/с), что делает решение пригодным для практического использования.

Сравнение `WireGost` с `ruWireGuard`

Сравнение с `ruWireGuard` наглядно демонстрирует неэффективность реализации высоконагруженных VPN-решений в пространстве пользователя (`userspace`) и использования языка программирования Go для тяжелых алгоритмов.

- **Пропускная способность:** перенос кода в ядро позволил увеличить скорость в **12 раз** (с 61 до 743 Мбит/с);
- **Стабильность UDP:** реализация на `Go` теряет более 98% `UDP`-пакетов при нагрузочном тестировании. Это происходит из-за того, что планировщик ОС и сборщик мусора (`GC`) языка `Go` не успевают обрабатывать прерывания и переключать контекст (`Kernel` $\leftrightarrow$ `User`) с необходимой

скоростью. Модуль ядра WireGost демонстрирует потери на уровне 13%, что является приемлемым показателем при полной загрузке CPU;

- **Задержки:** задержка в ruWireGuard (441 мс) делает его непригодным для VoIP и интерактивных приложений. WireGost обеспечивает задержку менее 0.1 мс.

Парадокс низкой задержки

Интересно отметить, что задержка под нагрузкой (Latency) у WireGost (0.083 мс) оказалась даже ниже, чем у оригинального WireGuard (0.098 мс). Это явление возможно объясняется эффектом *Bufferbloat*. Из-за сверхвысокой пропускной способности оригинального WireGuard сетевые буферы (qdisc) заполняются максимально плотно, создавая очереди. В случае с WireGost «узким местом» является CPU, а не сетевая очередь, поэтому пакеты, успевшие зашифроваться, отправляются в сеть практически мгновенно, не ожидая в переполненном буфере.

С другой стороны, если данное предположение ошибочно, то отличия в задержке являются незначительными и их можно списать на погрешность измерений.

3.6 Выводы по главе

В ходе выполнения раздела автором была решена задача программной реализации протокола WireGost в виде загружаемого модуля ядра Linux.

1. разработаны и интегрированы в подсистему Crypto API модули российских криптографических алгоритмов: хэш-функция «Стрибог» (с поддержкой HMAC), блочный шифр «Кузнечик» (в режиме MGM) и протокол выработки общего ключа VKO ГОСТ Р 34.10-2012;

2. модифицирована машина состояний протокола Noise и механизм рукопожатия для поддержки увеличенных размеров ключей (512 бит);

3. адаптирован интерфейс Netlink и разработана утилита wg_gost_cli

для управления интерфейсом, так как стандартные утилиты несовместимы с новыми размерами ключей;

4. проведенное нагрузочное тестирование показало, что разработанное решение обеспечивает максимальную пропускную способность (по UDP) до **862 Мбит/с**, что в **20 раз** превосходит существующие аналоги в пространстве пользователя, обеспечивая при этом надежную защиту данных в соответствии с отечественными стандартами.

Подтверждена корректность работы всех криптографических примитивов и протокола, в целом (тесты целостности файлов и успешное прохождение рукопожатий). В случае дальнейшей доработки разработанный модуль возможен для эксплуатации в средах с высокими требованиями к производительности.

4 Заключение

В ходе выполнения выпускной квалификационной работы было проведено исследование методов реализации виртуальных сетевых интерфейсов в ядре операционной системы Linux и успешно решена задача создания высокопроизводительного VPN-решения, использующего отечественные криптографические стандарты.

4.1 Текущие результаты

В рамках работы были достигнуты следующие результаты:

1. **Проведен анализ архитектуры ядра Linux:** изучены механизмы работы сетевого стека, подсистемы Crypto API и протокола Netlink. Обоснован выбор реализации в пространстве ядра (Kernel Space) как единственного способа обеспечения высокой производительности для высоконагруженных каналов связи;

2. **Разработан комплекс криптографических модулей:** реализованы и внедрены в ядро Linux алгоритмы ГОСТ Р 34.11-2012 («Стрибог»), ГОСТ Р 34.12-2015 («Кузнечик») в режиме MGM и протокол выработки

общего ключа VKO ГОСТ Р 34.10-2012. Все криптографические алгоритмы реализованы в виде модулей реализующих интерфейсы Linux Crypto API;

3. **Создан прототип VPN-интерфейса WireGost:** на базе архитектуры протокола WireGuard и фреймворка Noise Protocol Framework разработан модуль ядра, обеспечивающий построение защищенных тунелей с использованием отечественных криптографических алгоритмов;

4. **Разработан инструментарий управления:** создана утилита конфигурации wg_gost_cli, взаимодействующая с ядром через протокол Generic Netlink, что позволило обойти ограничения стандартных утилит WireGuard;

5. **Проведено нагрузочное тестирование:** экспериментально подтверждено, что перенос реализации ГОСТ-криптографии в пространство ядра обеспечивает существенный прирост производительности. Пропускная способность разработанного решения достигает в случае UDP **862 Мбит/с** и в случае TCP – **765 Мбит/с**, что в **20 раз** и **12 раз** соответственно превосходит существующие аналоги, работающие в пространстве пользователя (ruWireGuard-Go).

Сравнительный анализ показал, что, несмотря на отставание от оригинального WireGuard (обусловленное высокой вычислительной сложностью блочного шифра «Кузнечик» по сравнению с потоковым ChaCha20), разработанный модуль WireGost пригоден для практического использования в сетях с пропускной способностью до 1 Гбит/с.

4.2 Дальнейшее направление развития проекта

Основным направлением развития проекта является оптимизация криптографических примитивов с использованием векторных инструкций процессора (AVX/SSE) для повышения скорости шифрования, а также улучшение поддержки отечественной криптографии в ядре операционной системы Linux.

Более того, требуется дальнейшее исследование и разработка инструмента для конфигурации и управления модулем `WireGost`, так как текущий интерфейс не приспособлен для массового использования.

Разработанное программное обеспечение может служить основой для создания доверенных отечественных средств защиты информации, интегрируемых непосредственно в ядро операционной системы `Linux`.

Список литературы

1. *Bernstein D. J.*, *Lange T.*, *Niederhagen R.* Dual EC: A Standardized Back Door. — 2015. — URL: https://eprint.iacr.org/2015/767.
2. *Denis Kolegov*. Проект протокола Ru-WireGuard. — URL: https://github.com/bi-zone/ruwireguard-spec/tree/main.
3. *Donenfeld J. A.* WireGuard : Next Generation Kernel Network Tunnel. — 2020. — URL: https://www.wireguard.com/papers/wireguard.pdf.
4. Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012 / S. V. Smyshlyaev [и др.]. — 03.2016. — DOI: 10.17487/RFC7836. — URL: https://www.rfc-editor.org/info/rfc7836. RFC 7836.
5. ISO/IEC 7498-1 : 1994(E). — 1996.
6. *Kale C. J.*, *Socolofsky T. J.* TCP/IP tutorial. — 1991. — Янв. — DOI: 10.17487/RFC1180. — URL: https://www.rfc-editor.org/info/rfc1180.
7. *Karthikeyan Bhargavan and Gaëtan Leurent*. Sweet32: Birthday attacks on 64-bit block ciphers. — URL: https://sweet32.info/.
8. *Krawczyk H.*, *Bellare M.*, *Canetti R.* HMAC: Keyed-Hashing for Message Authentication. — 02.1997. — DOI: 10.17487/RFC2104. — URL: https://www.rfc-editor.org/info/rfc2104. RFC 2104.
9. *Mueller S.*, *Vasut M.* Linux Crypto API Documentation. — URL: https://docs.kernel.org/crypto/index.html#crypto-apihtml.
10. *The kernel development community*. Build system [Электронный ресурс] : Building External Modules. — 2025. — URL: https://docs.kernel.org/kbuild/modules.html (дата обр. 23.12.2025).
11. *The kernel development community*. Introduction to Netlink. — URL: https://docs.kernel.org/userspace-api/netlink/intro.html.
12. *The kernel development community*. Kernel modules. — URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html.

13. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : include/crypto.h. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/crypto.h (дата обр. 23.12.2025).

14. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : include/crypto/hash.h. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/crypto/hash.h (дата обр. 23.12.2025).

15. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : include/crypto/aead.h. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/crypto/aead.h (дата обр. 23.12.2025).

16. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : include/crypto/kpp.h. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/crypto/kpp.h (дата обр. 23.12.2025).

17. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : drivers/net/wireguard/noise.c. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireguard/noise.c (дата обр. 23.12.2025).

18. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : drivers/net/wireguard/send.c. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireguard/send.c (дата обр. 23.12.2025).

19. *The Linux Kernel Organization*. Linux Kernel Source Tree [Электронный ресурс] : drivers/net/wireguard/cookie.c. — 2025. — URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireguard/cookie.c (дата обр. 23.12.2025).

20. *Wu P.* Analysis of the WireGuard protocol. — 2019. — URL: https://research.tue.nl/files/130180306/Peter.Wu_Wireguard_thesis_final.pdf.

21. ГОСТ 28906-91 : ВЗАИМОСВЯЗЬ ОТКРЫТЫХ СИСТЕМ. БАЗОВАЯ ЭТАЛОННАЯ МОДЕЛЬ. — 1991. — URL: https://meganorm.ru/Data2/1/4294825/4294825816.pdf.

22. ГОСТ IEC 60050-732-2017 : Международный электротехнический словарь. Часть 732. Технологии компьютерных сетей. — 2024. — URL: https://meganorm.ru/Data/740/74056.pdf.

23. ГОСТ Р 1323565.1.026—2019 : Информационная технология. Криптографическая защита информации. Режимы работы блочных шифров, реализующие аутентифицированное шифрование. — 2019. — URL: https://meganorm.ru/Data2/1/4293727/4293727270.pdf.

24. ГОСТ Р 34.10-2012 : Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи. — 2012. — URL: https://meganorm.ru/Data2/1/4293788/4293788463.pdf.

25. ГОСТ Р 34.11-2012 : Информационная технология. Криптографическая защита информации. Функция хеширования. — 2012. — URL: https://meganorm.ru/Data2/1/4293788/4293788459.pdf.

26. ГОСТ Р 34.12-2015 : Информационная технология. Криптографическая защита информации. Блочные шифры. — 2015. — URL: https://meganorm.ru/Data2/1/4293762/4293762704.pdf.

27. ГОСТ Р 34.12-2018 : Информационная технология. Криптографическая защита информации. Блочные шифры. — 2018. — URL: https://meganorm.ru/Data2/1/4293732/4293732907.pdf.

28. ГОСТ Р 50922-2006 : Защита информации. Основные термины и определения. — 2008. — URL: https://meganorm.ru/Data2/1/4293836/4293836037.pdf.

29. ГОСТ Р 53531-2009 : ТЕЛЕВИДЕНИЕ ВЕЩАТЕЛЬНОЕ ЦИФРОВОЕ. ТРЕБОВАНИЯ К ЗАЩИТЕ ИНФОРМАЦИИ ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА В СЕТЯХ КАБЕЛЬНОГО И НАЗЕМНОГО ТЕЛЕВИЗИОННОГО ВЕЩАНИЯ. — 2009. — URL: https://meganorm.ru/Data2/1/4293819/4293819349.pdf.

30. Постановление Правительства РФ от 16.04.2012 N 313 : (ред. от 18.05.2017). — 2017. — URL: https://ivo.garant.ru/#/document/70164728.