

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

**Программные средства обеспечения унифицированного
интерфейса на основе протокола PKCS#11**

Студент

Свинарев Егор Сергеевич

Руководитель

Иванчуков А.Г.

Нормоконтролёр

Такташкин Д.В.

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой

Козлов А.Ю.

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

Попова Н.А.

Пенза, 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
А.Ю. Козлов
«20» декабря 2024 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Свинарев Егор Сергеевич гр. 21ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Программные средства обеспечения унифицированного интерфейса
на основе протокола PKCS#11

Тема утверждена приказом ПГУ № _____ от " _____ " _____ 20 _____

2. Руководитель работы к.т.н, доцент кафедры МОиПЭВМ Иванчуков А.Г.

3. Задание на работу (назначение разработки, исходные данные и т.п.)

а) Разработать программные компоненты, обеспечивающие унифицированный
интерфейс на основе протокола PKCS#11

б) Среда выполнения приложения: ОС Windows 10 и выше

Языковые средства разработки: C++, UML

Инструментальные средства разработки: VS Code, Visual Paradigm

в) Основные функции программного обеспечения:

1) Вычисление электронной подписи

2) Проверка электронной подписи

3) Вычисление хеш-значения

4) Настройка и конфигураций разрабатываемой подсистемы

4. Перечень подлежащих разработке вопросов

1) Анализ предметной области

2) Проектирование

3) Реализация

4) Контроль качества программного обеспечения

5) Оформление пояснительной записки

5. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	08.04.2025 – 14.04.2025	
2	Проектирование программных средств	20%	15.04.2025 – 25.04.2025	
3	Реализация программных средств	30%	26.04.2025 – 12.05.2025	
4	Тестирование программных средств	20%	13.05.2025 – 22.05.2025	
5	Оформление пояснительной записки	20%	23.05.2025 – 03.06.2025	

Дата выдачи задания «20» декабря 2024г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 135 страниц, 34 рисунка, 18 таблиц, 7 приложений.

PKCS#11, КРИПТОГРАФИЧЕСКИЙ ТОКЕН, HSM, RPC, APACHE THRIFT, МИКРОСЕРВИС, УНИФИКАЦИЯ ИНТЕРФЕЙСА, ГОСТ, UML, C++.

Объектом исследования являются программные средства, реализующие унифицированный интерфейс обращения к средствам электронной подписи различных производителей на основе протокола PKCS#11.

Цель работы – разработка и экспериментальная проверка программных средств обеспечения унифицированного интерфейса на основе протокола PKCS#11.

Средства исследования: анализ стандарта PKCS#11 и библиотек токенов, сравнительный, технический и функциональный анализ, UML-моделирование архитектуры, прототипирование на C++ с использованием Apache Thrift, модульное тестирование.

Среда выполнения экспериментов – Debian GNU/Linux 11, компилятор GCC 10, пакетный менеджер Conan, система сборки CMake, средства моделирования Visual Paradigm, среда разработки VS Code.

Результаты работы: создана подсистема обеспечения унифицированного интерфейса на основе протокола PKCS#11 для параллельного обращения к средствам электронной подписи от основной системы. Подтверждена инженерная эффективность решения через упрощение интеграции различных средств подписи через единый интерфейс.

					ПГУ 09.03.04 – 8BP211.22 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Программные средства обеспечения унифицированного интерфейса на основе протокола PKCS#11 Пояснительная записка	Лит.	Лист	Листов
Разраб.		Свинарев Е. С.						
Провер.		Иванчуков А.Г.					4	135
Н. Контр.		Такташкин Д.В,				Группа 21ВП1		
Утв.		Козлов А.Ю						

Содержание

Введение	6
1 Анализ предметной области	8
1.1 Электронная подпись в информационных системах.....	8
1.2 Унифицированные протоколы взаимодействия со средствами электронной подписи.....	8
1.2.1 CryptoAPI	8
1.2.2 PKCS#11	10
1.2.3 Обоснование выбора интерфейса	11
1.2.4 Обзор существующих решений	12
1.3 Проблемы интерфейса криптографических токенов	19
1.4 Постановка задачи на разработку.....	20
2 Анализ требований к системе	22
2.1 Функциональные требования	22
2.2 Нефункциональные требования	25
3 Проектирование архитектуры и интерфейсов	28
3.1 Архитектура системы	28
3.2 Компонент интерфейса криптографических операций	32
3.3 Компонент интерфейса управляющих операций	34
3.4 Компонент маршрутизации	40
3.5 Компонент кеширования обработчиков	43
3.6 Компоненты обработчиков криптографических и управляющих операций.....	44
3.7 Средства реализации.....	47
3.8 Сторонние зависимости.....	50
4 Реализация подсистемы унифицированного интерфейса.....	51
4.1 Интерфейс процедур удаленного вызова	51
4.2 Использование кодогенерации	53
4.3 Слойная схема Apache Thrift	54
4.4 Реализация бизнес-логики и обработчиков.....	55

4.5 Маршрутизация и управление обработчиками.....	60
4.6 Настройка и конфигурация микросервиса	62
4.7 Вспомогательные утилиты.....	64
5 Контроль стандартов качества программных решений	66
5.1 Функциональное тестирование.....	66
5.2 Функциональное тестирование средствами причинно-следственного графа и попарного тестирования	68
5.3 Модульное тестирование	72
5.4 Метрики качества программного обеспечения.....	76
5.4.1 Вычисление и определение метрики SLOC	76
5.4.2 Вычисление и определение метрики Холстеда	77
Заключение	81
Список использованных источников	83
Приложение А (справочное) Глоссарий	85
Приложение Б (рекомендуемое) Листинг примеров работы с интерфейсами криптографических токенов.....	89
Приложение В (обязательное) Исходный код приложения	92
Приложение Г (рекомендуемое) Диаграммы классов сгенерированного кода интерфейсов	111
Приложение Д (рекомендуемое) Диаграммы классов разработанного приложения	118
Приложение Е (рекомендуемое) Результаты функционального тестирования	124
Приложение Ж (рекомендуемое) Акт о внедрении	133

Введение

Цифровая трансформация бизнеса и государственного сектора обострила требования к надёжной криптографической защите данных: электронная коммерция, системы электронного документооборота и облачные сервисы массово используют электронную подпись. В зависимости от требований по производительности, безопасности и отказоустойчивости для выполнения операций электронной подписи могут применяться такие средства, как смарт-карты, USB-токены или HSM. Также ряд производителей предоставляют на рынок полностью программные решения, не использующие аппаратные составляющие. Универсальным интерфейсом для работы с такими средствами электронной подписи стал стандарт PKCS#11, позволяющий прикладному ПО выполнять операции подписи, шифрования и хеширования по единым стандартам.

Несмотря на формальную унификацию, развитие стандарта PKCS#11 и маркетинговая конкуренция привели к тому, что производители не стремятся обеспечить полное соответствие его требованиям. В результате при подключении средств электронной подписи нового поставщика велика вероятность столкнуться с несовместимостью на уровне программного обеспечения. Это, в свою очередь, приводит к увеличению затрат на интеграцию, тестирование и сопровождение инфраструктуры.

В данной работе предлагается решение задачи интеграции средств электронной подписи от различных производителей в информационную систему электронного документооборота. Основной целью является снижение затрат при переходе на ранее не поддерживаемые решения. Для этого разрабатывается подсистема, предоставляющая унифицированный интерфейс на основе протокола PKCS#11. Она должна обеспечивать информационной системе электронного документооборота единый способ обращения к средствам электронной подписи.

Для достижения цели требуется провести анализ предметной области, изучить существующие решения на рынке средств электронной подписи, а также сформулировать требования к разрабатываемой подсистеме. На основании предъявленных требований необходимо проработать архитектуру решения и выполнить проектирование и реализацию компонентов подсистемы. Корректность работы решения подтверждается на этапе тестирования.

В рамках работы применяются методы сравнительного, технического и функционального анализа, моделирование UML, а также экспериментальная разработка и нагрузочное тестирование прототипа в среде Debian.

1 Анализ предметной области

1.1 Электронная подпись в информационных системах

Электронная подпись является средством подтверждения авторства и неизменности электронных документов. Она обеспечивает возможность удостоверить личность автора документа и гарантировать, что документ не был изменен после подписания. Электронные подписи широко используются в системах электронного документооборота, финансовых транзакциях, а также при обмене сообщениями, требующими юридической значимости и защиты от подделок. На практике электронные подписи создаются с использованием специализированных аппаратных криптографических устройств, таких как HSM, USB-токены и смарт-карты. Такие устройства защищены от несанкционированного доступа и предоставляют криптографические операции через стандартизированные интерфейсы, упрощающие интеграцию и использование.

1.2 Унифицированные протоколы взаимодействия со средствами электронной подписи

Для обеспечения работы со средствами электронной подписи обычно используются стандартизованные протоколы и спецификации.

Наиболее распространенными среди них являются CryptoAPI и PKCS#11, которые получили широкое распространение в прикладных и системных решениях. Далее будут подробно рассмотрены их особенности и функциональные возможности

1.2.1 CryptoAPI

CryptoAPI представляет собой интерфейс прикладного программирования, разработанный корпорацией Microsoft для интеграции криптографических функций в приложения под операционной системой Windows [1]. Интерфейс предоставляет стандартизированный набор операций, включающий генерацию и

управление ключами, шифрование и дешифрование данных, создание и проверку цифровых подписей, а также вычисление криптографических хеш-функций и управление цифровыми сертификатами (таблица 1). CryptoAPI активно используется в разнообразных приложениях, требующих надежной защиты информации, таких как электронный документооборот, системы авторизации и аутентификации, защита электронной почты и веб-сервисов.

Преимуществами CryptoAPI являются простота использования и интеграции с приложениями под Windows, широкая поддержка различных алгоритмов и возможность использования централизованного хранилища сертификатов. Среди недостатков следует выделить привязку к конкретной платформе (Windows), ограниченные возможности по расширению функционала и возможные проблемы с производительностью в высоконагруженных приложениях и специализированных аппаратных средах.

Таблица 1 – Перечень основных функций, описываемых CryptoAPI

№	Категория	Функции	Описание
1	Инициализация	CryptAcquireContext	Открытие или создание контейнера ключей
2	Управление ключами	CryptGenKey	Генерация ассиметричных или симметричных ключей
		CryptImportKey	Импорт внешних ключей в контейнер
3	Криптографические операции	CryptSignHash	Подпись хеша документа
		CryptVerifySignature	Верификация цифровой подписи
		CryptEncrypt	Шифрование данных
		CryptDecrypt	Расшифровка данных
4	Хеширование	CryptCreateHash	Создание объекта хеширования
		CryptHashData	Обновление объекта хеша данными
		CryptGetHashParam	Получение итогового хеша или параметров
5	Шифрование произвольных данных	CryptProtectData	Шифрование данных с использованием DPAPI
		CryptUnprotectData	Дешифровка, восстановление исходного содержимого

Продолжение таблицы 1

№	Категория	Функции	Описание
6	Сертификаты	CertOpenStore	Открытие хранилища сертификатов
		CertCloseStore	Заккрытие хранилища сертификатов
		CertFindCertificateInStore	Поиск сертификатов по критериям
		CertGetCertificateContextProperty	Чтение свойств сертификата

В качестве примера использования стандарта CryptoAPI в программном коде на языке C++20 составлен листинг Б.1, демонстрирующий ключевые этапы работы с CryptoAPI: инициализацию контекста провайдера, создание и обновление объекта хеширования, вычисление подписи и корректное освобождение ресурсов.

1.2.2 PKCS#11

PKCS#11, также известный как Cryptoki, представляет собой универсальный прикладной программный интерфейс для взаимодействия с разнообразными криптографическими устройствами [2]. Стандарт предоставляет функции для генерации ключей, выполнения операций подписи, шифрования и дешифрования данных, управления криптографическими объектами и сертификатами (таблица 2). Основным преимуществом PKCS#11 является его платформонезависимый характер и способность работать с широким спектром устройств различных производителей, обеспечивая тем самым высокую степень совместимости и гибкости. Среди недостатков можно выделить относительную сложность первичной реализации и поддержки в связи с разнообразием специфических расширений и дополнений, которые производители интегрируют в свои устройства и программные библиотеки.

Таблица 2 – Перечень основных функций, описываемых PKCS#11

№	Категория	Функции	Описание
1	Инициализация	C_Initialize	Инициализирует библиотеку
		C_Finalize	Освобождает ресурсы библиотеки

Продолжение таблицы 2

№	Категория	Функции	Описание
2	Создание сессии	C_OpenSession	Открывает сессию с токеном в указанном слоте
		C_CloseSession	Закрывает ранее открытую сессию
3	Авторизация	C_Login	Аутентифицирует пользователя или администратора токена в сессии
		C_Logout	Завершает аутентификацию в заданной сессии
4	Управление объектами	C_CreateObject	Создает новый объект (ключ, сертификат, данные) в токене
		C_CopyObject	Копирует существующий объект с возможностью изменить набор атрибутов
		C_DestroyObject	Удаляет объект из токена
5	Криптографические операции	C_SignInit, C_Sign	Подготовка и выполнение операции подписи данных
		C_EncryptInit, C_Encrypt	Подготовка и выполнение операции хеширования данных
		C_DigestInit, C_Digest	Вычисление криптографического хеша данных

Вместе с перечнем синхронных криптографических операций, поддерживающих асимметричную и симметричную криптографию, стандарт регламентирует поддержку асинхронных операций: C_SignUpdate, C_SignFinal.

В качестве примера использования стандарта PKCS#11 в программном коде на языке C++20 составлен листинг Б.2, демонстрирующий ключевые этапы работы с PKCS#11: инициализация библиотеки, открытие и закрытие сессии, аутентификация пользователя, выполнение операции цифровой подписи и корректное завершение работы.

1.2.3 Обоснование выбора интерфейса

Для внутреннего потребителя – системы электронного документооборота – в данной работе в качестве основного интерфейса взаимодействия со средствами электронной подписи выбран PKCS#11. Это решение обусловлено его широкой распространенностью, высокой степенью универсальности и независимости от платформы. Поддержка различных криптографических устройств и алгоритмов делает PKCS#11 удобным выбором для создания гибких и масштабируемых сервисов.

В то же время проведенный анализ показал, что возможно расширение подсистемы посредством внедрения адаптера, обеспечивающего совместимость с решениями, использующими CryptoAPI. Такой подход позволит обеспечить поддержку более широкого круга криптографических провайдеров без жесткой привязки к конкретной платформе.

1.2.4 Обзор существующих решений

В отечественной инфраструктуре электронной подписи широко используются аппаратные модули безопасности (HSM). Эти устройства обеспечивают генерацию и защищённое хранение криптографических ключей, ускорение операций электронной подписи и шифрования, а также соответствие требованиям регуляторов. Общей чертой современных отечественных HSM является поддержка стандартизованного интерфейса PKCS#11, что обеспечивает совместимость с прикладными системами и удобную интеграцию.

1.2.4.1 CryptoPro CSP и аппаратный модуль CryptoPro HSM

CryptoPro CSP – это программный криптографический провайдер компании «КриптоПро», обеспечивающий поддержку российских алгоритмов электронной подписи и шифрования во многих программных платформах. CryptoPro CSP сертифицирован ФСБ России и может применяться для защиты конфиденциальной информации. Он поддерживает стандарт PKCS#11, что позволяет использовать его в приложениях через унифицированный интерфейс, а также включает модули для работы с API Microsoft CryptoAPI, CNG и Java JCA [3]. CryptoPro CSP обеспечивает реализацию отечественных криптоалгоритмов и при необходимости может работать и с международными алгоритмами. Для хранения ключей и выполнения операций CryptoPro CSP может взаимодействовать с внешними токенами и специализированными HSM.

CryptoPro HSM – аппаратно-программный комплекс от компании «КриптоПро», предназначенный для безопасного хранения ключей и ускоренного выполнения криптоопераций. Этот HSM выполнен в виде

серверного устройства с датчиками вскрытия корпуса и доверенной операционной системой. CryptoPro HSM сертифицирован по высоким классам защиты и обеспечивает защиту ключевой информации уровня KB2. В устройстве реализована многоуровневая система управления ключами, в которой активация HSM требует совместных действий нескольких администраторов для предотвращения несанкционированного доступа.

CryptoPro HSM работает под управлением защищённой ОС и предоставляет прикладным системам интерфейсы PKCS#11, CryptoAPI и JCA для доступа к криптофункциям. Это означает, что переход от программного провайдера к аппаратному модулю происходит бесшовно – приложения могут выполнять криптооперации через стандартные вызовы, не изменяя своей логики, но с использованием ключей внутри HSM. Пользовательские ключи никогда не покидают пределы HSM в открытом виде: для доступа к ним приложение проходит аутентификацию, а сами операции выполняются внутри защищённого модуля. CryptoPro HSM поддерживает разграничение ролей администраторов и ведёт журнал событий безопасности. Для управления и мониторинга предоставляются утилиты и API, входящие в клиентское ПО. В состав поставки входят также ключевые носители и считыватели для инициализации и администрирования HSM.

Основное назначение CryptoPro HSM – защита ключей электронной подписи удостоверяющих центров и других критичных узлов инфраструктуры электронной подписи. Благодаря высокой производительности модуль подходит для сервисов массового подписания документов, облачных сервисов ЭП, сервисов штампов времени и других. Например, в составе решения CryptoPro DSS (облачная электронная подпись) HSM используется как центральное хранилище ключей клиентов для удалённой подписи документов. Высокий класс защиты позволяет использовать CryptoPro HSM в банковских системах, например, для выполнения требований Единой биометрической системы – ключи для подписи биометрических данных хранятся и управляются в HSM. Дополнительно CryptoPro HSM может применяться для подписи кода.

Преимуществами решения являются отечественная сертификация, высокая производительность, поддержка полного перечня российских алгоритмов и совместимость с популярными серверными платформами. К ограничениям можно отнести относительно высокую стоимость и требования к размещению, что делает CryptoPro HSM оправданным для крупных организаций и поставщиков услуг, но избыточным для мелких систем.

1.2.4.2 ViPNet CSP и аппаратный модуль ViPNet HSM

ViPNet CSP – это программный криптографический провайдер, разработанный компанией «ИнфоТеКС». Он аналогичен по своим функциям CryptoPro CSP, обеспечивая реализацию российских стандартов криптографии в операционных системах и приложениях. ViPNet CSP сертифицирован ФСБ России и допущен к защите информации ограниченного доступа, что позволяет использовать его в государственных и корпоративных системах электронного документооборота [4]. Данный провайдер полностью поддерживает алгоритмы семейства ГОСТ. ViPNet CSP, как и решение от CryptoPro, ориентирован на использование отечественных криптосредств, однако для совместимости может работать и с иностранными алгоритмами при наличии соответствующих плагинов.

Архитектурно ViPNet CSP встраивается в инфраструктуру PKI, являясь обязательным компонентом клиентского ПО ViPNet и обеспечивая приложениям доступ к криптографическим операциям. ViPNet CSP функционирует на платформах Windows и Linux, предоставляя стандартные интерфейсы CryptoAPI и PKCS#11 для приложений [5]. Для работы с аппаратными носителями ключевой информации ViPNet CSP может использовать библиотеки PKCS#11, предоставляемые производителями токенов. Например, интеграция с USB-токенами Rutoken ЭЦП осуществляется установкой библиотеки rtPKCS11ECP.dll – после этого ViPNet CSP может выполнять генерацию ключевой пары и хранение закрытого ключа непосредственно на токене [6]. Аналогичным образом обеспечивается

совместимость с токенами JaCarta и другими носителями – через соответствующие PKCS#11-драйверы. Таким образом, ViPNet CSP поддерживает широкий спектр российских аппаратных средств криптографической защиты информации.

ViPNet CSP ценится за реализацию отечественных алгоритмов и соответствие требованиям регуляторов. Он часто используется в составе комплексных решений Infotecs – например, для обеспечения криптографических функций рабочего места в системе электронного документооборота, при авторизации и аутентификации пользователей в защищённых системах, для шифрования каналов связи ViPNet Coordinator и других. Благодаря поддержке PKCS#11 и CryptoAPI, данный провайдер может быть интегрирован практически в любые приложения, требующие электронную подпись или шифрование по ГОСТ. Ограничения ViPNet CSP схожи с таковыми у CryptoPro CSP – программная реализация уступает аппаратным HSM по скорости операций и устойчивости к взлому. Поэтому в ответственных узлах ViPNet CSP рекомендуется использовать совместно с аппаратными модулями, выступая в роли промежуточного слоя между приложением и устройством.

ViPNet HSM – отечественный аппаратный HSM, разработанный компанией «ИнфоТеКС». По концепции он близок к рассмотренному выше решению от CryptoPro, то есть представляет собой высокопроизводительную программно-аппаратную платформу для выполнения криптографических операций и хранения ключей в защищённой среде. ViPNet HSM выполнен на базе серверного оборудования с доверенной загрузкой и датчиками защищённости, имеет встроенный сенсор вскрытия корпуса и аппаратный генератор случайных чисел. Устройство работает под управлением собственной безопасной операционной среды и поддерживает удалённое администрирование. В результате сертификации ФСБ России ViPNet HSM получил сертификаты, подтверждающие соответствие классам СКЗИ КВ и средств ЭП класса КВ2. Это означает, что модуль может использоваться как для защиты информации, так и

для квалифицированной электронной подписи в юридически значимых системах.

ViPNet HSM реализует полный набор отечественных алгоритмов для электронной подписи и шифрования. Основные функции включают генерацию ключей шифрования, шифрование данных и имитовставку, создание и проверку ключей электронной подписи, формирование и проверку электронной подписи, и вычисление хеш-функции. Поддерживается также выработка общих ключей по протоколу Диффи-Хеллмана на эллиптических кривых и вычисление HMAC по ГОСТ Р 34.11-2012. Таким образом, ViPNet HSM покрывает все потребности инфраструктуры электронной подписи и шифрования в рамках российских стандартов. Начиная с версии 3.3, платформа получила поддержку иностранных криптоалгоритмов для симметричного шифрования AES, формирования и проверки электронной подписи по стандарту DSS, функций SHA-256/384/512, HMAC по NIST SP 800-38B и других. Это позволяет использовать ViPNet HSM в интегрированных системах, требующих сочетания ГОСТ и мировых алгоритмов.

ViPNet HSM представляет собой универсальный криптомодуль, к которому могут подключаться различные прикладные сервисы по стандартному API. Для взаимодействия с приложениями предусмотрен интерфейс PKCS#11 – HSM выступает как криптографический токен с поддержкой необходимых функций. Поддерживаются клиенты под управлением Windows и Linux, что покрывает основные серверные платформы. ViPNet HSM может работать в составе отказоустойчивого кластера: предусмотрена возможность горизонтального масштабирования с балансировкой нагрузки между несколькими модулями, что актуально для высоконагруженных систем. Особенностью ViPNet HSM является возможность встраивания сторонних приложений и сервисов в его защищенную среду. По сути, Infotecs позиционирует HSM как платформу, в которой разработчики могут создавать новые средства криптографической защиты или электронные подписи, которые будут работать внутри HSM в изолированной доверенной среде. Это расширяет

сценарии использования – например, на базе ViPNet HSM реализован модуль ViPNet PKI Service. Администрирование устройства осуществляется группой операторов с разграничением ролей, а все критические события журналируются. Для локальной настройки на корпусе устройства может присутствовать сенсорный экран, а для удалённого – веб-интерфейс управления. Применение защищённой ОС и аппаратного контроля целостности гарантирует отсутствие вмешательства в работу HSM извне.

ViPNet HSM, будучи отечественным решением, закрывает потребность в аппаратных модулях безопасности для органов власти, силовых структур и коммерческих организаций, где требуется соответствие российским криптостандартам и наличие сертификатов ФСБ. Он используется в инфраструктуре электронного правительства, в системах юридически значимого электронного документооборота, для централизованного хранения ключей квалифицированной ЭП.

В финансовом секторе ViPNet HSM может применяться в роли платежного HSM из-за реализованной поддержки операций с картами «Мир», EMV-протоколов Visa/MasterCard, а также генерации и проверки PIN-кодов. ViPNet HSM также находит применение в сервисах удалённой электронной подписи, так как он позволяет безопасно хранить пользовательские ключи и выполнять операции ЭП по запросу через защищённые каналы.

К преимуществам решения относят высокую степень безопасности, гибкость, масштабируемость, а также широкую поддержку отечественных алгоритмов, включая новейшие стандарты.

Ограничения во многом схожи с другими аппаратными HSM – высокая стоимость внедрения и обслуживания, необходимость квалифицированного администрирования, интеграция требует применения SDK или стандартов, что может потребовать доработки ПО.

В целом, ViPNet HSM конкурирует с CryptoPro HSM в сегменте высокозащищённых модулей для инфраструктуры открытых ключей.

1.2.4.3 **Аппаратные токены Rutoken**

Rutoken – семейство аппаратных USB-ключей от российской компании «Актив», широко применяемых для хранения ключей электронной подписи и аутентификации пользователей. Токены Rutoken поддерживают интерфейс PKCS#11 версии 2.20, что позволяет использовать их во всех стандартных приложениях, поддерживающих PKCS#11 [7]. Кроме того, для токенов Rutoken разработаны собственные криптопровайдеры и драйверы, облегчающие их использование в Windows и других ОС.

Существует несколько моделей Rutoken, различающихся функциональностью и уровнем защиты. Наиболее простые модели, такие как Rutoken Lite или Rutoken S, служат защищёнными носителями ключевой информации: они имеют встроенный защищённый микроконтроллер и память для хранения ключевых контейнеров, однако могут не выполнять криптографические операции самостоятельно. Например, Rutoken S предназначен для безопасного хранения ключей ГОСТ и сертифицирован ФСТЭК России для использования в государственных ИС. Более продвинутые модели – Rutoken ЭЦП 2.0 и Rutoken ЭЦП 3.0 – содержат на борту полноценный криптографический модуль, способный выполнять генерацию ключевых пар, вычисление электронной подписи и шифрование прямо внутри токена. Такие устройства имеют сертификаты ФСБ России по классам КС1 или КС2, позволяющие их использование для юридически значимой электронной подписи. Например, ряд токенов Rutoken ЭЦП сертифицирован ФСБ как средства электронной подписи класса КС2, что отвечает требованиям к усиленной квалифицированной подписи.

Криптографические возможности токенов включают аппаратную реализацию алгоритмов ГОСТ. В частности, Rutoken ЭЦП 2.0/3.0 поддерживает электронную подпись по ГОСТ Р 34.10-2012, выработку общего ключа по протоколу Диффи-Хеллмана, выработку имитовставки, а также классическое симметричное шифрование по ГОСТ 28147-89. Для работы с зарубежными

алгоритмами используются программные реализации, однако последние поколения токенов начали поддерживать и некоторые иностранные алгоритмы.

Для разработки приложений с поддержкой Rutoken производитель предоставляет PKCS#11 библиотеку rtPKCS11 и документацию по функциям PKCS#11, реализованным токеном [8]. Через этот интерфейс приложение может выполнять генерацию ключа внутри токена, получать открытый ключ, вычислять ЭП либо расшифровывать данные с использованием закрытого ключа. Все чувствительные операции выполняются на самом устройстве, что исключает возможность кражи ключа даже при компрометации компьютера. Rutoken совместим с популярными российскими программными продуктами CryptoPro CSP и ViPNet CSP при установке драйверов, которые позволяют использовать его как криптографический провайдер для ГОСТ.

Преимуществами Rutoken являются портативность и автономность, так как пользователю достаточно носить с собой небольшой USB-ключ, который можно подключить к любому компьютеру для совершения подписи. Токены обеспечивают высокий уровень защиты из-за аппаратных реализаций ГОСТ, устойчивых к большинству программных атак, а попытки извлечения ключа из микросхемы требуют сложного лабораторного воздействия. Ограничения включают сравнительно низкую производительность, где выполнение одной операции подписи ГОСТ на токене занимает порядка долей секунды, поэтому для массового подписания USB-токены не подходят. Объем памяти накладывает ограничения на количество одновременных ключей/сертификатов, которые можно хранить. Таким образом, Rutoken оптимален для персонального применения.

1.3 Проблемы интерфейса криптографических токенов

Разнообразие аппаратных устройств и программных библиотек, реализующих интерфейс PKCS#11, неизбежно ведет к обширному множеству расширений, специфичных для конкретного производителя – каждая компания добавляет собственные механизмы, атрибуты и объекты, призванные закрыть

пробелы в стандарте интерфейса. В качестве пробелов может выступать поддержка новых алгоритмов, тонких политик доступа или асинхронных операций. В результате таких решений приложение, которое успешно работало с одной программной библиотекой, при переключении на другую сталкивается с несовместимостью: одни и те же атрибуты трактуются по-разному, специфичные для поставщика коды ошибок не документированы, нужные механизмы отсутствуют или имеют другой идентификатор. Вместе с этим ситуацию осложняет одновременное существование нескольких версий спецификации, от чего требуется наличие условных ветвей кода и динамический запрос адресов функций.

Описанные выше проблемы ведут к увеличению трудоемкости при разработке и тестировании, затрудняют миграции между устройствами, вносят риск ограничения переносимости между решениями разных поставщиков и вынуждают организации создавать промежуточные абстракции или ограничиваться самым общим подмножеством возможностей PKCS#11, жертвуя производительностью, стабильностью и функциональностью.

Все это показывает, что необходимо универсальное решение, которое позволит работать с разными токенами без доработки под каждого производителя. Таким решением может стать единый программный шлюз, который будет скрывать различия между библиотеками и предоставлять одинаковый интерфейс для всех.

1.4 Постановка задачи на разработку

Целью разработки является создание программных средств, обеспечивающих унифицированный программный интерфейс для взаимодействия приложений с криптографическими устройствами и токенами на основе протокола PKCS#11. Разрабатываемая система должна реализовывать интерфейс и обеспечивать выполнение ключевых криптографических операций с возможностью конфигурирования параметров подсистемы и организации удалённого взаимодействия.

В процессе выполнения работы необходимо решить следующие задачи:

- произвести анализ функциональных и нефункциональных требований к разрабатываемой системе;
- спроектировать архитектуру системы и ее компонент с учетом заданных функциональных и нефункциональных требований;
- определить средства реализации на основе представленной архитектуры;
- выполнить реализацию программных модулей для поддержки выполнения криптографических операций;
- разработать механизмы конфигурации параметром подсистемы и средства удаленного взаимодействия;
- протестировать и отладить работоспособность подсистемы для обеспечения надежности и корректности поставляемых подсистемой интерфейсов.

Результаты ключевых этапов построения программных средств необходимо разместить в приложениях.

2 Анализ требований к системе

2.1 Функциональные требования

Для взаимодействия с функциями средств электронной подписи и конфигурацией системы пользователь должен обладать ролью информационных средств электронного документооборота, который может использовать специализированные интерфейсы управления и выполнения криптографических операций. Управляемыми объектами являются криптографические операции, настройки подсистемы и подключения.

Криптографические операции включают:

- вычисление электронной подписи;
- проверка электронной подписи;
- вычисление хеш-значения.

Пользователь должен иметь возможность выбора конкретного типа криптографической операции перед ее выполнением, что позволяет гибко управлять функционалом системы в зависимости от задачи.

При настройке службы криптографии разработчик может задавать параметры поставщиков операций, профили использования, алгоритмы и проверять корректность настроек, включая корректность сертификатов и устройства подписи.

Вместе с этим, пользователь должен иметь возможность настройки взаимодействия с удаленной службой, включая задание IP-адреса, порта и проверку корректности параметров подключения. В процессе работы система должна проводить строгую валидацию введенных данных и корректно обрабатывать ошибки и исключения.

Согласно требованиям, определенным в предыдущих абзацах, составлена диаграмма вариантов использования [9], представленная на рисунке 1 и иллюстрирующая основные варианты использования системы.

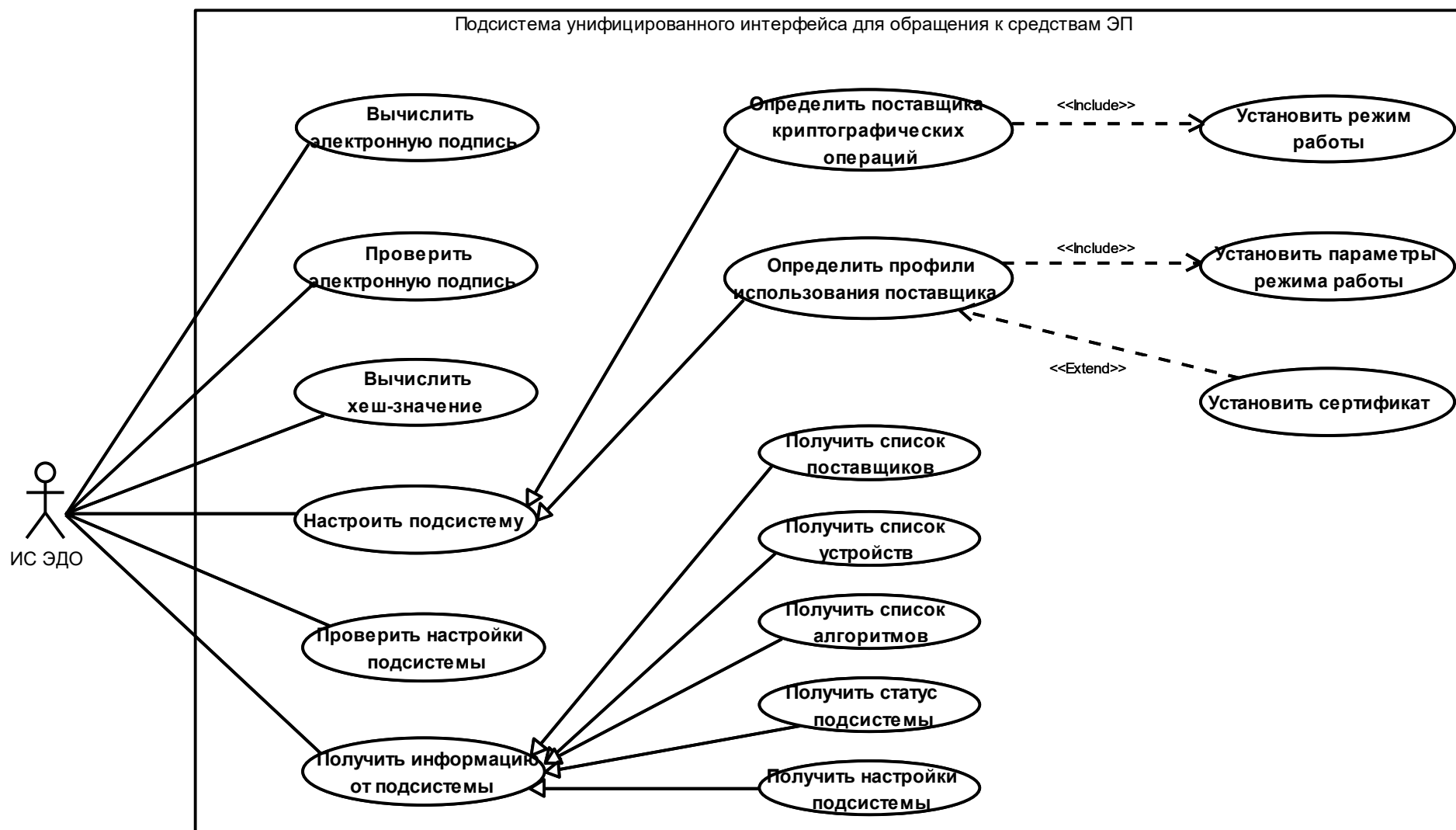


Рисунок 1 – Диаграмма вариантов использования

Вариант использования «Подписать данные» (рисунок 2) позволяет информационной системе выполнить криптографическую операцию подписи данных через унифицированный интерфейс.

Прецедент: подписать данные
ID: 1
Краткое описание: ИС ЭДО инициирует операцию подписи переданных данных
Главные актеры: ИС ЭДО
Второстепенные актеры: нет
Предусловия: подсистема настроена и доступна
Основной поток: <ol style="list-style-type: none"> 1. ИС ЭДО формирует данные для подписи 2. ИС ЭДО направляет запрос на выполнение операции подписи 3. Подсистема выполняет операцию и возвращает результат
Постусловие: данные подписаны

Рисунок 2 – Спецификация варианта использования «Подписать данные»

Вариант использования «Проверить подпись» (рисунок 3) позволяет информационной системе убедиться в подлинности подписанных данных.

Прецедент: проверить подпись
ID: 2
Краткое описание: ИС ЭДО инициирует проверку подписи
Главные актеры: ИС ЭДО
Второстепенные актеры: нет
Предусловия: подсистема доступна, переданы данные и подпись
Основной поток: <ol style="list-style-type: none"> 1. ИС ЭДО передаёт данные и подпись 2. Подсистема проверяет подпись 3. Система возвращает результат проверки
Постусловие: подпись проверена

Рисунок 3 – Спецификация варианта использования «Проверить подпись»

Вариант использования «Вычислить хеш» (рисунок 4) позволяет информационной системе получить хеш переданных данных.

Прецедент: вычислить хеш
ID: 3
Краткое описание: ИС ЭДО инициирует вычисление хеш-значения
Главные актеры: ИС ЭДО
Второстепенные актеры: нет
Предусловия: подсистема доступна
Основной поток: <ol style="list-style-type: none"> 1. ИС ЭДО направляет данные 2. Подсистема выполняет операцию хеширования 3. Система возвращает результат
Постусловие: хеш-значение получено

Рисунок 4 – Спецификация варианта использования «Вычислить хеш»

Вариант использования «Настроить подсистему» (рисунок 5) позволяет информационной системе задать параметры поставщика криптографических операций и режим работы.

Прецедент: настроить подсистему
ID: 4
Краткое описание: ИС ЭДО задаёт параметры подсистемы
Главные актеры: ИС ЭДО
Второстепенные актеры: нет
Предусловия: подсистема доступна
Основной поток: <ol style="list-style-type: none"> 1. ИС ЭДО определяет поставщика криптографических операций 2. ИС ЭДО задаёт режим работы 3. ИС ЭДО определяет профиль использования
Постусловие: подсистема настроена и готова к работе

Рисунок 5 – Спецификация варианта использования «Настроить взаимодействие с удаленной службой»

Вариант использования «Получить информацию от подсистемы» (рисунок 6) позволяет информационной системе получить сведения о текущих параметрах и состоянии подсистемы.

Прецедент: получить информацию от подсистемы
ID: 5
Краткое описание: ИС ЭДО получает данные о состоянии и настройках подсистемы
Главные актеры: ИС ЭДО
Второстепенные актеры: нет
Предусловия: подсистема доступна
Основной поток: <ol style="list-style-type: none"> 1. ИС ЭДО запрашивает список поставщиков 2. ИС ЭДО запрашивает список устройств 3. ИС ЭДО запрашивает список алгоритмов 4. ИС ЭДО получает настройки и статус подсистемы
Постусловие: сведения о подсистеме получены

Рисунок 6 – Спецификация варианта использования «Получить информацию о конфигурации запуска»

Составленная диаграмма вариантов использования описывает множество функций, обязательных для реализации разрабатываемой подсистемой.

2.2 Нефункциональные требования

Системные нефункциональные требования определяют характеристики и качества подсистемы, которые не связаны с функциональностью, но которые

оказывают существенное влияние, если рассматривать эксплуатацию в реальных условиях.

Перечень системных нефункциональных требований, определяемых для разрабатываемых программных средств, представлен в таблице 3.

Таблица 3 – Системные нефункциональные требования

№	Требование	Описание	Критерии завершения
1	Производительность	Система должна обеспечивать высокую производительность и низкое время отклика на запросы	Среднее время отклика для операции подписи или вычисления хеша не должно превышать 100 миллисекунд
2	Надежность, доступность и восстанавливаемость	Система должна обеспечивать высокий уровень доступности и минимальное время на восстанавливаемость	<div>Общий процент доступности подсистемы не должен быть менее 99.9% в год</div> <div>Время восстановления подсистемы после прерывания работоспособности не должно превышать 5 минут</div>
3	Совместимость	Система должна быть совместима с распространенными операционными системами и криптографическими устройствами	<div>Поддержка операционных систем семейства Linux</div> <div>Совместимость с криптографическими токенами и библиотеками российских производителей, реализующими интерфейс PKCS#11</div>
4	Безопасность	Система должна обеспечивать безопасность обработки данных и защиту от внешних угроз	<div>Отсутствие известных уязвимостей в используемых зависимостях</div> <div>Минимизация привилегий операционной системы, требуемых для работы</div>

Продолжение таблицы 3

№	Требование	Описание	Критерии завершения
5	Конфигурируемость и сопровождаемость	Система должна быть конфигурируема и сопровождается	Конфигурация подсистемы задается внешним файлом в формате JSON
			Информационные сообщения структурированы и удобны для анализа
6	Переносимость	Система должны быть переносима между различными окружениями	Развертывание микросервиса в средах контейнеризации
			Отсутствие жестких зависимостей от аппаратных или программных средств
7	Диагностируемость	Система должна предоставлять средства мониторинга состояния и диагностики исправностей	Реализация процедуры удаленного вызова для определения состояния сервиса
			Составление логируемых сообщений, множество которых доступно в потоке стандартного вывода
8	Документированность	Исходный код и сопровождающая документация должны быть ясными и удобными для сопровождения и расширения	Техническая документация по развертыванию, конфигурированию и эксплуатации микросервиса
			Исходный код следует стандарту форматирования и сопровождается пояснительными комментариями для нетривиальных технических решений

Описанные выше системные нефункциональные требования определяют взгляд на проектирование, не относящийся к предметной области, влияют на ряд структурных аспектов приложения, а также играют решающую или весьма важную роль в успехе разрабатываемой системы.

3 Проектирование архитектуры и интерфейсов

3.1 Архитектура системы

В соответствии с предъявленными функциональными и нефункциональными требованиями, в качестве архитектурного стиля был выбран модульный монолит. Такой выбор обусловлен множеством следующих причин:

- модульность – подсистема реализует строго ограниченный набор криптографических функций, что упрощает поддержку, тестирование и последующую доработку;
- масштабируемость – при необходимости для разрабатываемой подсистемы доступно развертывание в виде нескольких экземпляров для повышения показателей общей производительности;
- гибкость внедрения – подсистема может быть развернута как отдельно, так и в составе более сложных решений, взаимодействуя с другими компонентами через стандартизированный интерфейс RPC;
- изоляция зависимостей – каждое приложение может использовать собственную версию сторонних библиотек и настроек среды исполнения, не имея влияния на другие компоненты.

Для реализации интерфейсов взаимодействия с информационной системой планируется использовать клиент-серверную архитектуру. Такая архитектура обеспечивает разделение функциональности между серверной частью, которая обрабатывает основные запросы и бизнес-логику, и клиентской, которая взаимодействует с серверной частью по установленным протоколам обмена данными.

Для организации взаимодействия между компонентами системы выбран механизм удаленного вызова процедур (RPC, Remote Procedure Call). RPC позволяет производить вызов функций и процедур, которые находятся на удаленном сервере так, как если бы они выполнялись локально. Такой подход

упрощает реализацию распределенных приложений, снижает сложность интеграции разрабатываемой системы и абстрагирует детали сетевого взаимодействия.

В качестве основного и единственного инструмента для реализации RPC выбран фреймворк Apache Thrift. Фреймворк предоставляет многоязыковое окружение для создания кроссплатформенных приложений с поддержкой множества языков программирования, среди которых есть C++. Thrift позволяет описывать интерфейсы сервисных приложений и структуры данных при помощи собственного языка описания интерфейса (IDL, Interface Definition Language). После составления описания интерфейса приложения конкретный компилятор, поставляемый с фреймворком Apache Thrift, позволяет произвести автоматическую генерацию кода для серверной и клиентской стороны на выбранном языке [10]. Такой выбор определен следующими ключевыми факторами:

- эффективная передача данных между компонентами системы;
- поддержка широкого набора языков программирования;
- гибкая настройка уровней транспорта и протокола;
- доступный и удобочитаемый вариант кодогенерации;
- простой уровень интеграции в процесс разработки приложения.

Для демонстрации внутренней архитектуры подсистемы в рамках исполняемого файла представлена диаграмма компонентов (рисунок 7).

На диаграмме отражены ключевые элементы взаимодействия разрабатываемой подсистемы: реализация интерфейсов служебных и криптографических операций, а также взаимодействие с поставщиками криптографического оборудования.

Архитектура системы предусматривает возможность расширения путем подключения дополнительных криптографических провайдеров через описание соответствующих режимов работы в конфигурационном файле и интерфейсе служебных операций.

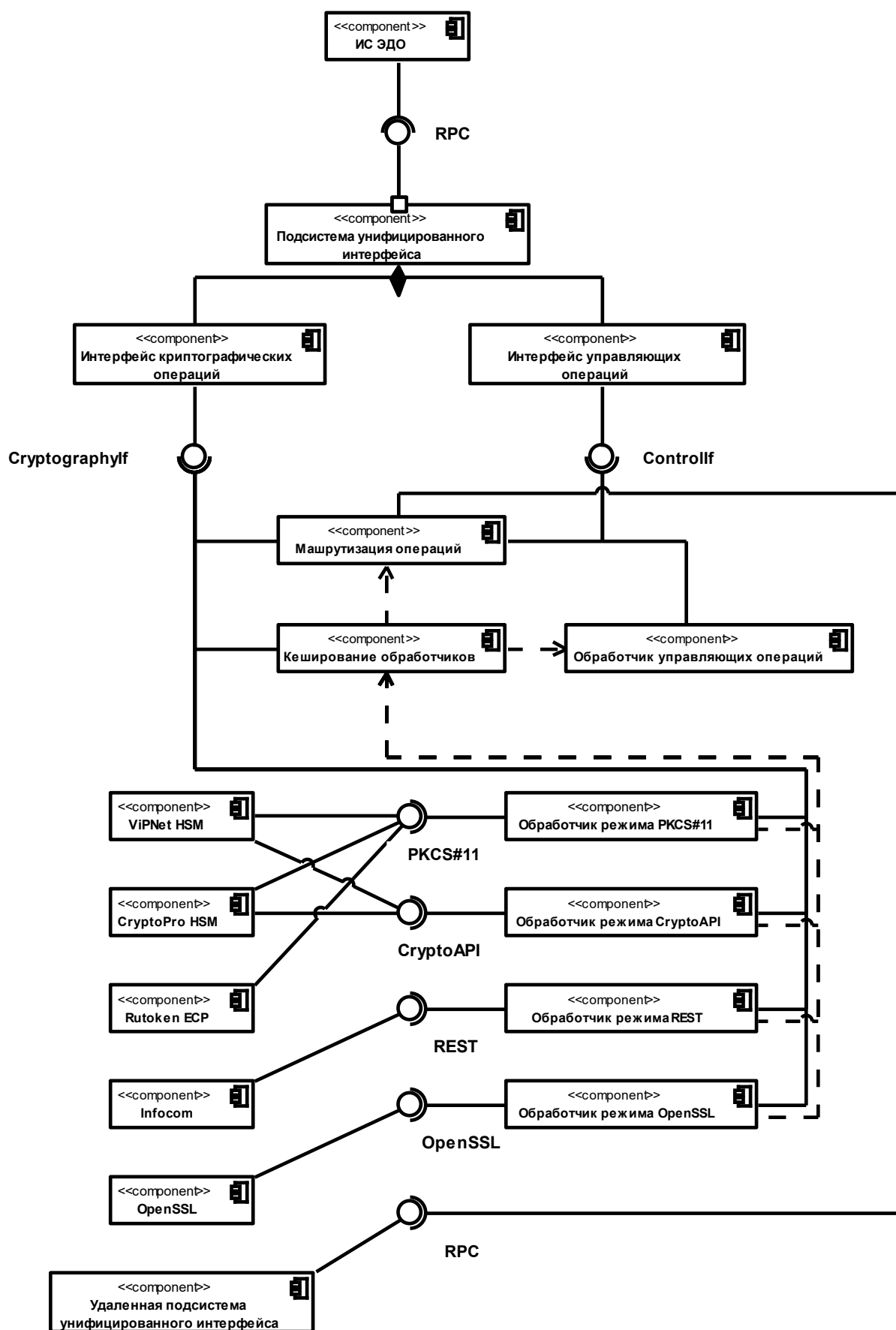


Рисунок 7 – Диаграмма компонентов

С целью демонстрации размещения разрабатываемой подсистемы в различных средах представлена диаграмма развертывания (рисунок 8).

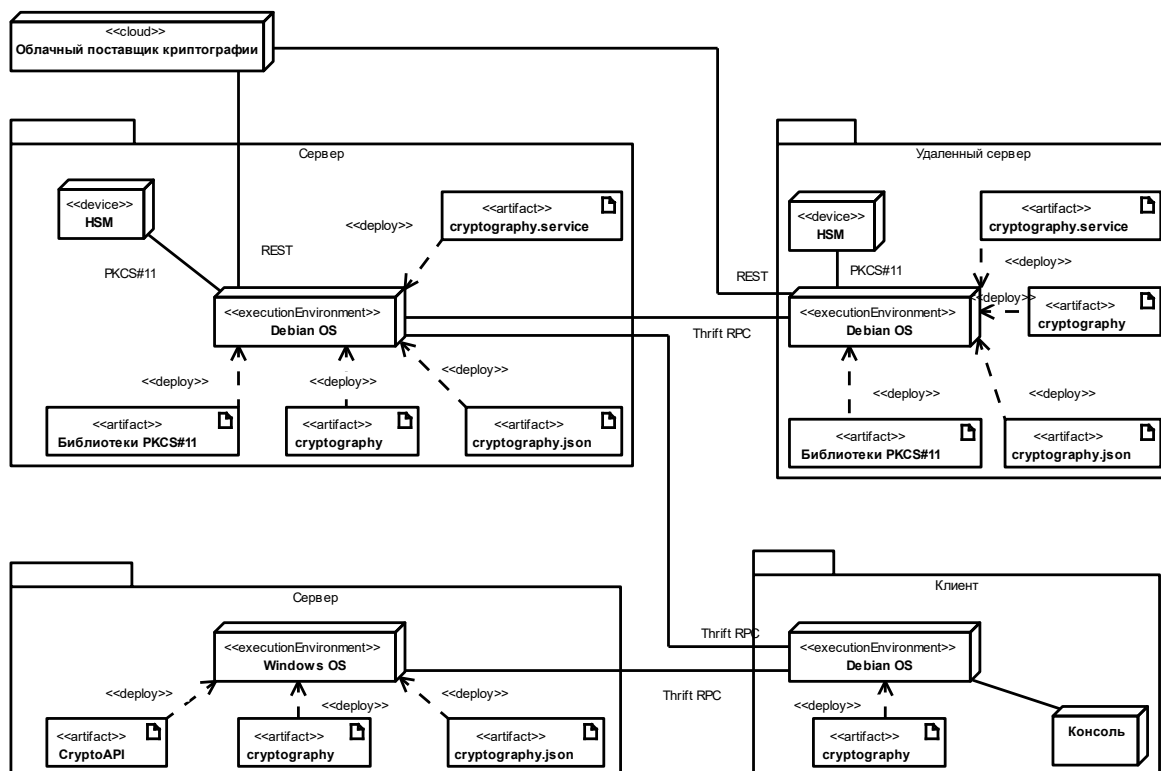


Рисунок 8 – Диаграмма развертывания

На ней показано, как служба криптографии может функционировать в нескольких режимах в зависимости от выбранного криптографического провайдера и окружения выполнения:

- локальный режим с использованием библиотек PKCS#11 и аппаратного модуля HSM;
- локальный режим с использованием CryptoAPI;
- удаленный режим, в котором разрабатываемый микросервис работает на внешнем сервере, доступном через Thrift RPC;
- интеграция с облачным криптографическим провайдером через интерфейс REST.

Во всех сценариях поведение разрабатываемой подсистемы управляется с помощью интерфейса служебных операций и средствами конфигурационного файла, в котором определяются настройки провайдеров, адреса для работы конечных точек, доступные для использования алгоритмы и другие опции.

3.2 Компонент интерфейса криптографических операций

Анализ функциональных требований показал необходимость предоставления удалённого доступа к основным криптографическим функциям устройства электронной подписи. К таким функциям относятся криптографические операции: электронная подпись данных, проверка электронной подписи и вычисление криптографического хеш-значения. Эти операции должны быть доступны через унифицированный интерфейс для различных криптографических устройств, что позволит системе гибко управлять функциональностью в зависимости от задач. Помимо этого, требования предусматривают строгую проверку корректности входных данных и обработку ошибок при выполнении операций, чтобы обеспечить надёжность взаимодействия с устройствами подписи. Учитывая данные потребности, был спроектирован отдельный интерфейс сервиса для криптографических операций, определенный в файле `cryptography.thrift` (приложение В.1), предназначенный для выполнения стандартных операций: подписи, хеширования и проверки подписи. Такой подход отвечает требованию унификации работы с разнородными токенами и HSM, скрывая различия реализации и предоставляя единый набор функций для всех устройств.

Структура интерфейса криптографических операций была спроектирована в отдельном IDL-файле `cryptography.thrift`, что обеспечивает логическую изоляцию криптографического функционала. В этом файле описаны необходимые пространства имён и структуры данных, а также объявлен сервис `Cryptography` с перечнем доступных процедур. Такое решение обеспечивает ясность и самостоятельность интерфейса, так как файл содержит все необходимые определения для работы с криптографическими операциями: структуры запросов и ответов, перечисления, синонимы типов и исключения. Разделение интерфейса на отдельный модуль `cryptography` подчёркивает его назначение и упрощает поддержку. При необходимости, изменения, связанные

непосредственно с криптооперациями, вносятся в одном месте, не затрагивая другие части системы.

В соответствии с требованиями, сервис Cryptography включает набор удалённых процедур для основных криптографических действий. Каждая процедура имеет чётко определённый набор параметров и возвращаемых результатов, что повышает типобезопасность и удобство использования интерфейса. Перечень основных операций следующий:

- Sign – процедура создания электронной подписи для переданных данных, который принимает структуру SignRequest с необходимыми параметрами и возвращает структуру SignResponse с полученной подписью;

- Verify – процедура проверки электронной подписи, где в VerifyRequest передаются хеш проверяемых данных, сама подпись и информация об открытом ключе, а результатом является булево значение в VerifyResponse, указывающее на достоверность подписи;

- Digest – процедура вычисления криптографического хеша сообщения, которая принимает DigestRequest, содержащий исходные данные и идентификатор алгоритма хеширования, а возвращает DigestResponse, включающий вычисленное хеш-значение.

Полный перечень процедур, определяемых интерфейсом криптографических операций, представлен на таблице 4.

Таблица 4 – Перечень процедур интерфейса криптографических операций

Процедура	Описание	Параметры	Результат
Sign	Создание подписи	Профиль режима работы, хеш сообщения	Подпись для сообщения
Verify	Проверка подписи	Профиль режима работы, хеш сообщения, подпись, описание открытого ключа	Результат проверки подписи
Digest	Вычисление хеша сообщения	Профиль режима работы, сообщение, OID алгоритма хеширования	Хеш сообщения

Продолжение таблицы 4

Процедура	Описание	Параметры	Результат
GetSignAlgorithm	Получение используемого алгоритма подписи	Профиль режима работы	Представление алгоритма подписи
GetDigestAlgorithm	Получение используемого алгоритма хеширования	Профиль режима работы	Представление алгоритма хеширования
GetCertificate	Получение используемого сертификата	Профиль режима работы	Бинарные данные используемого сертификата и других сертификатов в цепочке
GetStatus	Получение статуса службы	-	Статус службы

Проектирование интерфейса криптографических операций осуществлялось с учётом принципов читаемости, расширяемости и простоты интеграции. Разделение интерфейса на самостоятельный сервис Cryptography повысило его когерентность за счет группировки профильных криптографических функций, благодаря чему интерфейс получился интуитивно понятным и не перегруженным лишними методами. Такое логическое группирование упрощает интеграцию для внешних систем, которые видят компактный набор операций, прямо связанных с криптографией.

Немаловажно, что в интерфейсе реализован унифицированный подход к обработке ошибок через исключения Apache Thrift. Все методы объявлены с возможностью возбуждения `CryptographyException`, что позволяет централизованно обрабатывать сбои и информировать вызывающую сторону о проблемах. Так, вместо возврата кодов ошибок, которые нужно обрабатывать вручную, клиент получает стандартизованное исключение с сообщением, что повышает надёжность и упрощает отладку при интеграции.

3.3 Компонент интерфейса управляющих операций

Помимо криптографических функций, система должна предоставлять средства управления сервисом криптографии и его настройками. Согласно требованиям, пользователь системы должен иметь возможность

конфигурировать подсистему: задавать или изменять параметры криптографического провайдера, определять профили использования и проверять корректность этих настроек перед их применением. В сценариях эксплуатации также предусмотрено получение информации о текущем состоянии сервиса криптографии – например, перечень доступных провайдеров, подключённых устройств и поддерживаемых алгоритмов, а также выгрузка активных настроек и статуса работы сервиса. Дополнительно, система должна поддерживать удалённый режим: требуется возможность настроить подключение к удалённому сервису криптографии, указав IP-адрес и порт, проверив корректность соединения, а при необходимости – отключиться от удалённого сервиса. Эти возможности вынесены в отдельный интерфейс управляющих операций, определенных в файле `control.thrift` (приложение В.2), который дополняет криптоинтерфейс и содержит процедуры по применению настроек, их валидации, управлению режимами локальной и удаленной работы и получению сведений о состоянии подсистемы. Разделение функциональности на два интерфейса – криптографический и управляющий – вызвано требованием логической изоляции: криптографические операции и операции администрирования системы относятся к разным категориям, поэтому для удобства сопровождения и использования они представлены разными сервисами удалённого вызова.

Описание интерфейса управляющих операций вынесено в отдельный файл для чёткого разграничения административного функционала. Внутри этого файла определён сервис управления, разделённый на две логические части: локальные операции и дополнение для удалённого режима. Для реализации такого разделения в Thrift выбран механизм наследования сервисов, где базовый сервис `ControlLocal` содержит методы для локального управления, а сервис `Control` расширяет его, добавляя процедуры для управления удалённым подключением. Таким образом, в итоговом сервисе `Control` объединены оба набора методов, и клиент может работать с единым интерфейсом управления, при этом внутренняя реализация отличает локальные вызовы от удалённых.

При проектировании структур данных и пространств имён учитывалось повторное использование определений из интерфейса криптографических операций. Чтобы избежать дублирования и обеспечить согласованность, в интерфейс управления импортированы необходимые типы из криптографического интерфейса. Например, структура `Status` для представления статуса службы и структура `Algorithm` для описания алгоритмов. Помимо этого, определены собственные структуры для описания конфигурации. Так, поставщик криптографических операций с полями наименования, описания и режима работы определен в структуре `Provider`, объединение для описания режимов работы криптографического провайдера представлено в `Regime`, а структура для профиля использования, включающая алгоритм хеширования по умолчанию и данные сертификата, описана структурой `Profile`. Эти определения позволяют задать иерархическую модель настроек, описывающую этапы от выбора провайдера и режима его работы до отдельных параметров. По аналогии введена структура `Settings` как контейнер всех настроек сервиса, включая выбранного поставщика и набор профилей, а также `RemoteSettings` для параметров удалённого подключения, где указывается используемый адрес и порты.

Сервис `Control`, вместе с унаследованными методами `ControlLocal`, предоставляет обширный набор процедур для управления сервисом криптографии.

Группа процедур управления локальными настройками выглядит следующим образом:

- `SetSettings` – применяет новую конфигурацию для службы криптографии, где в качестве аргумента принимает структуру `SetSettingsRequest`, содержащую желаемые настройки сервиса;
- `CheckSettings` – выполняет проверку корректности заданных настроек без их сохранения, принимая `CheckSettingsRequest` с проверяемыми

настройками Settings, и возвращая структуру CheckSettingsResponse с прогнозируемым статусом системы в случае применения этих настроек;

- GetSettings – предоставляет текущие активные настройки сервиса, возвращая структуру GetSettingsResponse с заполненным объектом Settings;

- GetProviders – возвращает список доступных поставщиков криптографических операций;

- GetDevices – возвращает список обнаруженных устройств электронной подписи для текущего криптографического провайдера, обёрнутый в GetDevicesResponse;

- GetAlgorithms – возвращает перечень криптографических алгоритмов, поддерживаемых текущим провайдером, разделённых по категориям – алгоритмы хеширования и алгоритмы электронной подписи.

Группа процедур управления удалённым подключением представлена в таком виде:

- SetRemoteSettings – задаёт параметры для подключения к удалённому сервису криптографии, принимая структуру SetRemoteSettingsRequest с необходимыми сетевыми параметрами и выполняя настройку соединения;

- CheckRemoteSettings – проверяет корректность параметров удалённого подключения через параметр CheckRemoteSettingsRequest, который включает предполагаемые настройки удалённого подключения RemoteSettings;

- CloseRemoteConnection – разрывает ранее установленное подключение к удалённой службе и переводит систему в локальный режим работы;

- GetRemoteSettings – возвращает текущие сохранённые настройки удалённого подключения;

- GetRemoteStatus – предоставляет информацию о состоянии удалённого подключения через структуру GetRemoteStatusResponse,

включающую метку времени удалённых настроек, флаг, активировано ли удалённое взаимодействие, и флаг доступности удалённого сервиса.

Полный перечень процедур, определяемых интерфейсом управляющих операций, представлен на таблице 5.

Таблица 5 – Перечень процедур интерфейса управляющих операций

Процедура	Описание	Параметры	Результат
SetSettings	Применить настройки	Настройки службы криптографии	-
CheckSettings	Выполнить проверку настроек без применения	Настройки службы криптографии	Статус, который возможен, если настройки корректны
GetSettings	Получить активные настройки	-	Активные настройки
GetProviders	Получить список поставщиков криптографических операций	-	Поставщики криптографических операций
GetDevices	Получить список доступных устройств подписи	-	Перечень устройств подписи
GetAlgorithms	Получить список поддерживаемых алгоритмов	-	Перечень алгоритмов подписи и хеширования
GetStatus	Получить статус службы	-	Состояние службы
SetRemoteSettings	Установить настройки подключения к удаленной службе	Настройки подключения к удаленной службе	-
CheckRemoteSettings	Проверить настройки подключения к удаленной службе	Настройки подключения к удаленной службе	-
ClosetRemoteConnection	Разъединить подключение к удаленной службе	-	-
GetRemoteSettings	Получить настройки подключения к удаленной службе	-	Настройки подключения к удаленной службе
GetRemoteStatus	Получить статус удаленного подключения	-	Включен ли режим удаленного подключения, доступна ли удаленная служба

Все методы интерфейса управления, как и в случае криптографических операций, используют механизм исключений Apache Thrift для сигнализации об ошибках. В определениях методов через ключевое слово указано, что каждая функция может генерировать импортированное исключение `CryptographyException`. Это означает унифицированную обработку непредвиденных ситуаций, так как во всех случаях клиент получит структурированное уведомление об ошибке.

При проектировании интерфейса управляющих операций ключевым ориентиром было обеспечение гибкости настройки и контроля за сервисом криптографии при сохранении простоты и надёжности взаимодействия. Разделение методов на локальные и удалённые через базовый сервис и расширение позволило добиться логической группировки функций и повторного использования кода. Локальные операции сосредоточены на настройке текущего сервиса, тогда как удалённые – на управлении подключением. Однако для клиента они представлены единым сервисом `Control`, что упрощает модель взаимодействия, ведь нет необходимости отдельно обращаться к разным интерфейсам в коде. Такой дизайн соответствует принципу инкапсуляции деталей реализации. Переключение между локальным и удалённым режимом происходит через вызов `SetRemoteSettings` и изменение внутреннего состояния, после чего остальные методы начинают работать с удалённым сервисом автоматически.

Все необходимые управляющие функции, вытекающие из сценариев использования, нашли отражение в интерфейсе. Например, требование возможности выбора провайдера и профиля реализовано через комбинацию вызовов получения списка поставщиков, `GetProviders`, и применение выбранного поставщика и связанных параметров профиля через `SetSettings`. Требование проверки корректности конфигурации выполнено в виде отдельной процедуры `CheckSettings`, чтобы администратор мог протестировать настройки перед активацией. Необходимость получения информации о системе реализована методами `GetDevices`, `GetAlgorithms`, `GetSettings`, `GetStatus`. Согласно сценарию

«получить информацию от подсистемы» внешняя система последовательно вызывает эти функции и получает все нужные сведения. Для поддержки сценария подключения к удалённому криптосервису в интерфейс добавлены парные методы `SetRemoteSettings/CloseRemoteConnection` и сопутствующие функции проверки и чтения статуса, что закрывает соответствующее функциональное требование.

Принцип расширяемости был реализован за счёт продуманной структуры данных. К примеру, использование объединений `Regime` и профилей `Profile` позволяет легко добавлять новые режимы работы криптографического провайдера. Достаточно определить новую структуру параметров и добавить поле в `Regime` как `optional`, сохранив обратную совместимость. Добавление нового типа устройства или алгоритма также не потребует изменения существующих функций – они возвращают списки из типов `Device` и `Algorithm`, к которым можно добавить новые поля или значения перечислений. Это согласуется с общей архитектурой системы, позволяющей подключать новые криптографические модули через изменение конфигурации и интерфейса управляющих операций.

Наконец, важным проектным решением было обеспечивать надёжность и безопасность операций управления. Все методы проверяют входные параметры и используют исключения для отчетливой сигнализации об ошибках. Кроме того, разделение интерфейсов по разным сетевым портам позволяет при необходимости разграничить права доступа. Например, одни клиенты могут получать доступ только к криптооперациям, но не к административным функциям. Это способствует общей безопасности решения и соответствует принципам минимизации полномочий.

3.4 Компонент маршрутизации

Компонент маршрутизации предназначен для распределения поступающих вызовов между локальной реализацией криптографического сервиса и удалённым сервисом. Его задачей является направление вызовов RPC-

интерфейса либо на локальную обработку, либо на удалённый сервис, в зависимости от текущих настроек. Такой подход позволяет поддерживать сценарии, в которых криптографические операции могут выполняться локально или делегироваться удалённой системе, обеспечивая поддержку удаленного режима работы.

На рисунке 9 представлена диаграмма классов компонента маршрутизации.

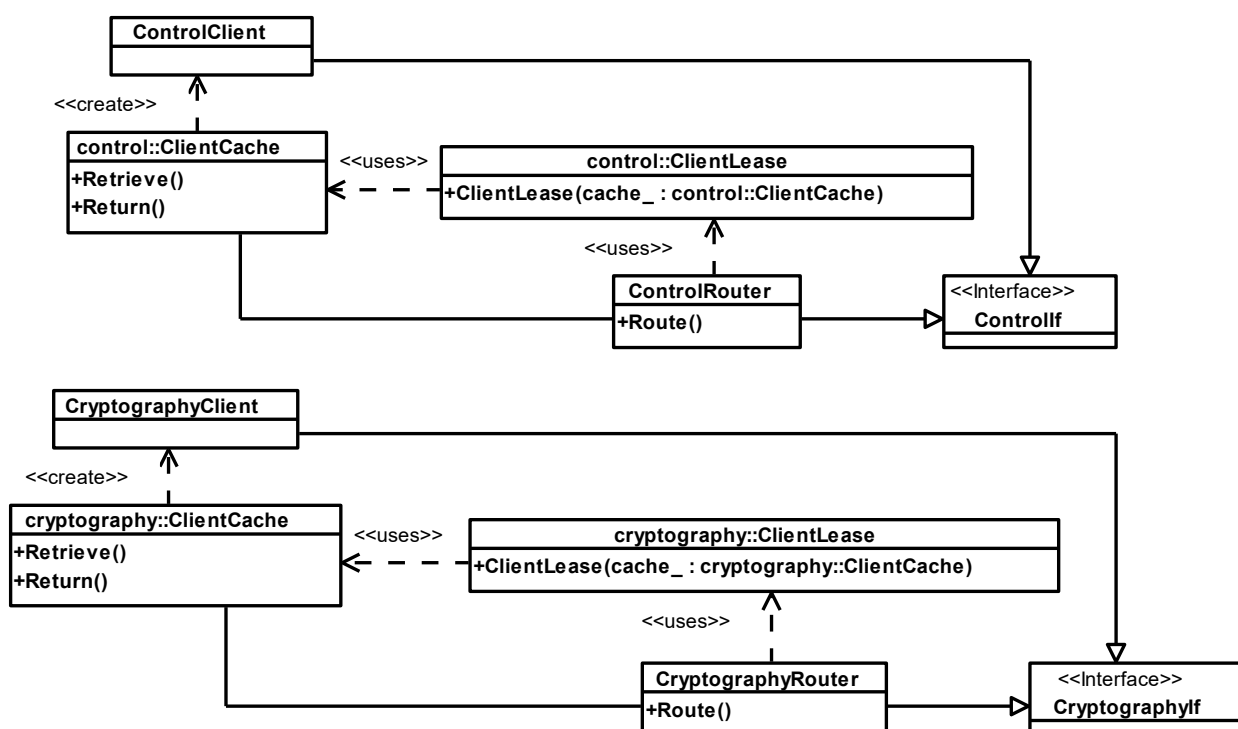


Рисунок 9 – Диаграмма классов компонента маршрутизации

Ключевыми классами здесь являются **CryptographyRouter** и **ControlRouter**. Оба класса реализуют соответствующие интерфейсы сервиса криптографических операций и сервиса управляющих операций и служат в качестве обработчиков верхнего уровня, инкапсулирующих логику маршрутизации. **CryptographyRouter** отвечает за маршрутизацию вызовов криптографических операций, а **ControlRouter** – за маршрутизацию вызовов операций управления сервисом. Метод `Route` в классе **CryptographyRouter** реализует основную логику маршрутизации. Он проверяет, настроено ли удалённое взаимодействие, и, в зависимости от этого, либо перенаправляет

вызов на локальный обработчик, либо инициирует удалённый вызов. Аналогично, класс `ControlRouter` реализует маршрутизацию по тому же принципу, что и `CryptographyRouter`, но предназначен для управляющего интерфейса.

Оба класса интегрируются в инфраструктуру Apache Thrift в качестве обработчиков соответствующих сервисов, поэтому остальной код прозрачно работает с ними, не зная о наличии маршрутизации.

В компоненте маршрутизации активно задействован идиоматический паттерн C++ RAII. Специализированные объекты, возвращаемые методом маршрутизации, `cryptography::ClientLease` и `control::ClientLease`, реализуют RAII для автоматического управления ресурсами, такими как сетевые соединения, реализуемые классами `CryptographyClient` и `ControlClient`. Это обеспечивает безопасность и освобождение ресурсов даже в случаях исключений.

Упрощённый сценарий обработки криптографической операции подписи в удалённом режиме с участием компонента маршрутизации выглядит следующим образом:

- Apache Thrift передаёт вызов объекту `CryptographyRouter`;
- в методе `Sign` класса `CryptographyRouter` внутренняя реализация вызывает функцию `Route`, передавая ей параметры вызова;
- внутри `Route` выполняется проверка: настроен ли удалённый криптосервис для выполнения операций подписи;
- `CryptographyRouter` через механизм кеширования извлекает объект клиента, обернутого в RAII-класс для автоматического возврата в кеш при смерти объекта, установив соединение при необходимости;
- `CryptographyRouter` вызывает метод `Sign` у удалённого обработчика, отправляя запрос в удалённый криптографический сервис;
- после получения результата `CryptographyRouter` возвращает его вызывающему клиенту.

Независимо от того, локальный или удаленный режим работы настроен сейчас, для внешнего клиента работа сервиса выглядит одинаково – детали маршрутизации скрыты внутри компонента.

3.5 Компонент кеширования обработчиков

Компонент кеширования обработчиков предназначен для эффективного управления объектами-обработчиками вызовов. В контексте данного проекта под обработчиками понимаются объекты, исполняющие бизнес-логику для RPC-вызовов. Кеширующий компонент решает задачу избегания повторного создания обработчика при множественных вызовах и обеспечивает повторное использование и потокобезопасность доступа для разработчиков. Таким образом удастся повысить производительность микросервиса и достичь более эффективного использования ресурсов.

На рисунке 10 представлена диаграмма классов компонента кеширования обработчиков.

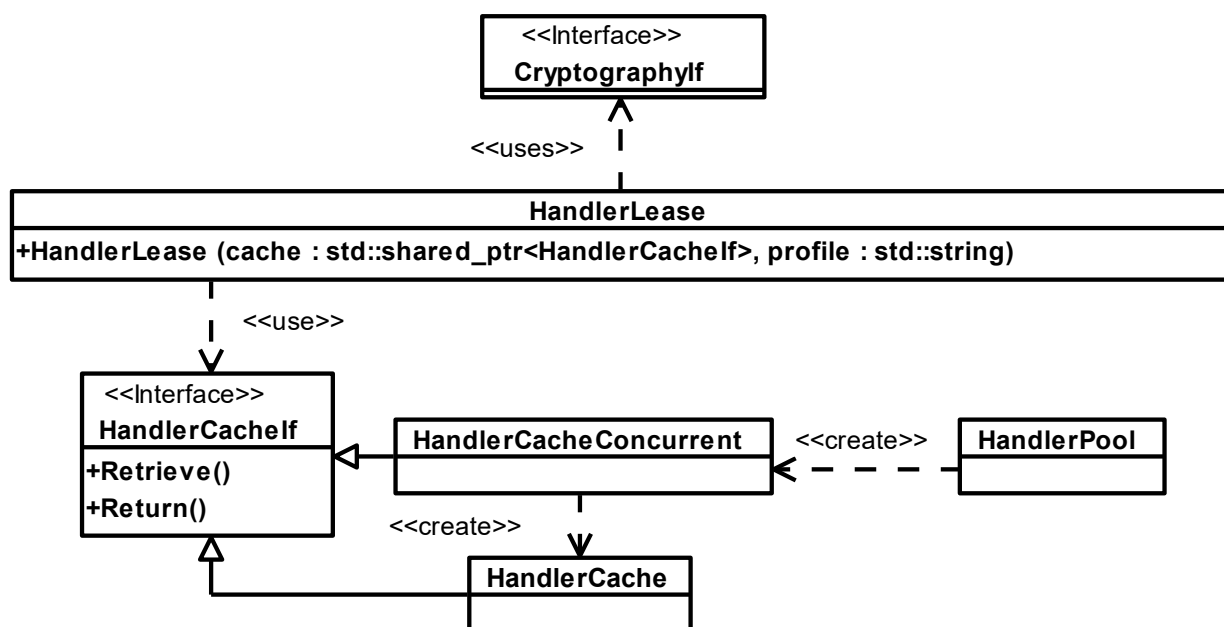


Рисунок 10 – Диаграмма классов компонента кеширования обработчиков

В основе диаграммы находится структура данных для хранения доступных обработчиков и набор классов-обёрток, обеспечивающих удобный интерфейс и

потокобезопасность. Базовым контейнером выступает неупорядоченный ассоциативный массив, допускающий повторяющиеся ключи. Ключом в этой структуре служит имя профиля, относительно которого создан обработчик вызова, а значением – экземпляр обработчика. Под профилем в данном контексте понимается набор настроек для использования поставщика.

Над этим контейнером реализован класс `HandlerCache`, инкапсулирующий доступ к структуре хранения. Диаграмма показывает, что от `HandlerCache` наследуется класс `HandlerCacheConcurrent`, добавляющий механизмы синхронизации для многопоточного доступа. Также представлены классы `HandlerLease` и `HandlerPool`, связанные с кэшированием: `HandlerLease` обеспечивает удобный RAII-механизм для работы с обработчиками, а `HandlerPool` реализует пул обработчиков, совмещённый с интерфейсом сервиса. Связи между этими классами таковы, что `HandlerPool` использует внутри себя `HandlerCacheConcurrent` и `HandlerLease` для управления коллекцией обработчиков, а `HandlerCacheConcurrent`, в свою очередь, реализует интерфейс, определённый `HandlerCache`. Эта иерархия организована для разделения ответственности: `HandlerCache` определяет абстракцию кеша, `HandlerCacheConcurrent` отвечает за потокобезопасность, `HandlerLease` за RAII-управление единичным ресурсом, а `HandlerPool` за масштабирование на уровень пула и интеграцию с остальным приложением

3.6 Компоненты обработчиков криптографических и управляющих операций

Обработчики криптографических и управляющих операций реализуют бизнес-логику сервиса. Они выполняют непосредственные действия при поступлении RPC-вызовов, будь то криптографическая функция или административная задача. Эти классы находятся на нижнем уровне иерархии обработчиков. Маршрутизатор направляет вызов на конкретный обработчик, который выполняет требуемую операцию, возможно взаимодействуя с криптографическим провайдером или библиотекой. Таким образом, назначение

данных компонентов заключается в обеспечении корректного выполнения функциональных требований системы. Криптографические операции должны выполняться с использованием требуемых алгоритмов и устройств, а управляющие операции должны надёжно менять состояние сервиса в ответ на команды управления.

На рисунке 11 представлена диаграмма классов обработчика криптографических операций.

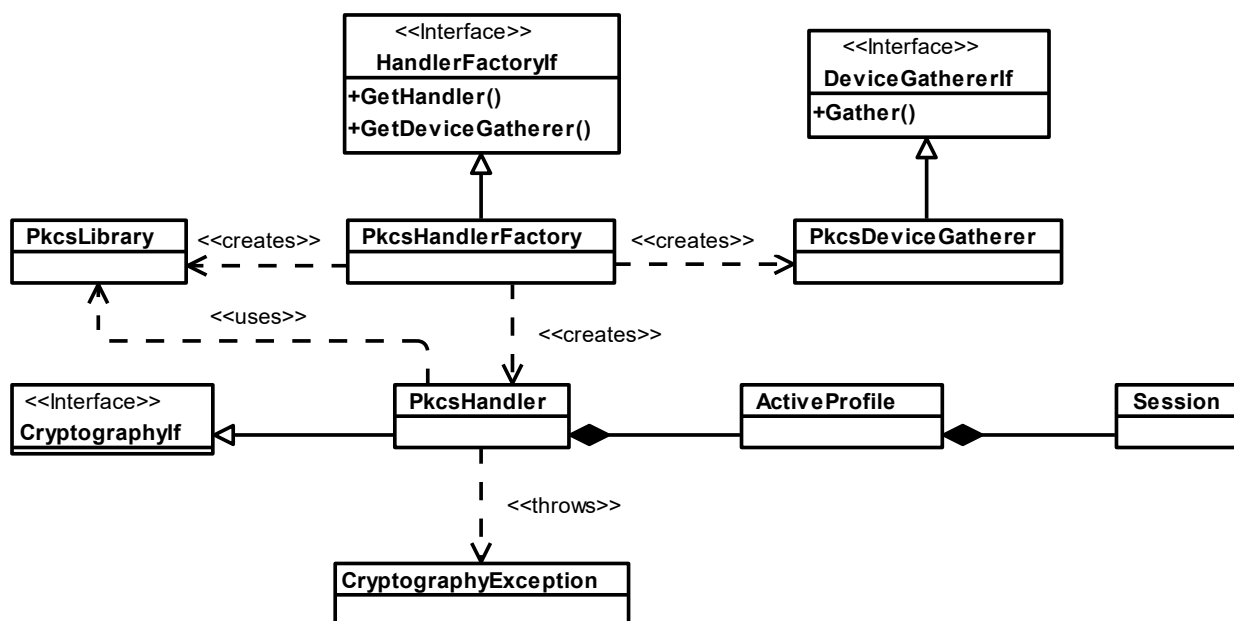


Рисунок 11 – Диаграмма классов обработчика криптографических операций

На диаграмме классов обработчика криптографических операций отражены классы, реализующие интерфейс **CryptographyIf** и связанные вспомогательные компоненты.

Реализация интерфейса криптографических операций определена в классе **PkcsHandler**. Название **PkcsHandler** отражает, что данный обработчик рассчитан на работу через PKCS#11. При инициализации **PkcsHandler** копирует объект **ActiveProfile**. **ActiveProfile** содержит состояние активного криптографического профиля, подготовленного на этапе инициализации фабрики обработчиков. Методы **PkcsHandler** при выполнении криптоопераций обращаются к объекту сессии из **ActiveProfile**. Так, метод подписи вызывает функции PKCS#11 для

создания цифровой подписи, используя ранее открытую сессию и выбранный секретный ключ. Таким образом, PkcsHandler инкапсулирует всю логику работы с внешним криптографическим провайдером.

Фабричным классом, отвечающим за подготовку и создание объектов-обработчиков криптографического сервиса, является PkcsHandlerFactory. Он реализует интерфейс HandlerFactoryIf, предоставляя методы для создания обработчика вызовов и получения объекта, определяющего доступные устройства подписи. PkcsHandlerFactory выполняет инициализацию криптографического провайдера при инстанцировании. В частности, PkcsHandlerFactory загружает динамическую библиотеку PKCS#11, инициализирует её, находит подходящий токен, открывает сессию и выбирает нужный ключ. Поскольку все обработчики ссылаются на один общий контекст, не требуется повторно выполнять тяжёлую инициализацию для каждого. Все они разделяют сессию и настройки, полученные фабрикой. PkcsHandlerFactory тем самым реализует паттерн Factory. Помимо создания обработчиков, PkcsHandlerFactory отвечает за предоставление объекта для поиска доступных криптографических устройств. Это нужно, чтобы по запросу клиента можно было узнать, какие устройства доступны для использования.

Для обнаружения и сбора информации о подключённых устройствах электронной подписи, связанных с PKCS#11, используется класс PkcsDeviceGatherer. Он реализует интерфейс DeviceGathererIf, в котором объявлен один метод, при вызове которого происходит сканирование доступных устройств.

Класс используется фабрикой, когда нужно предоставить пользователю сведения о том, с какими устройствами может работать сервис. В соответствии с принципами SOLID, функциональность по сбору информации о устройствах отделена от PkcsHandlerFactory именно в этот класс, чтобы фабрика отвечала только за создание обработчиков, а не за обнаружение устройств.

На рисунке 12 представлена диаграмма классов обработчика управляющих операций.

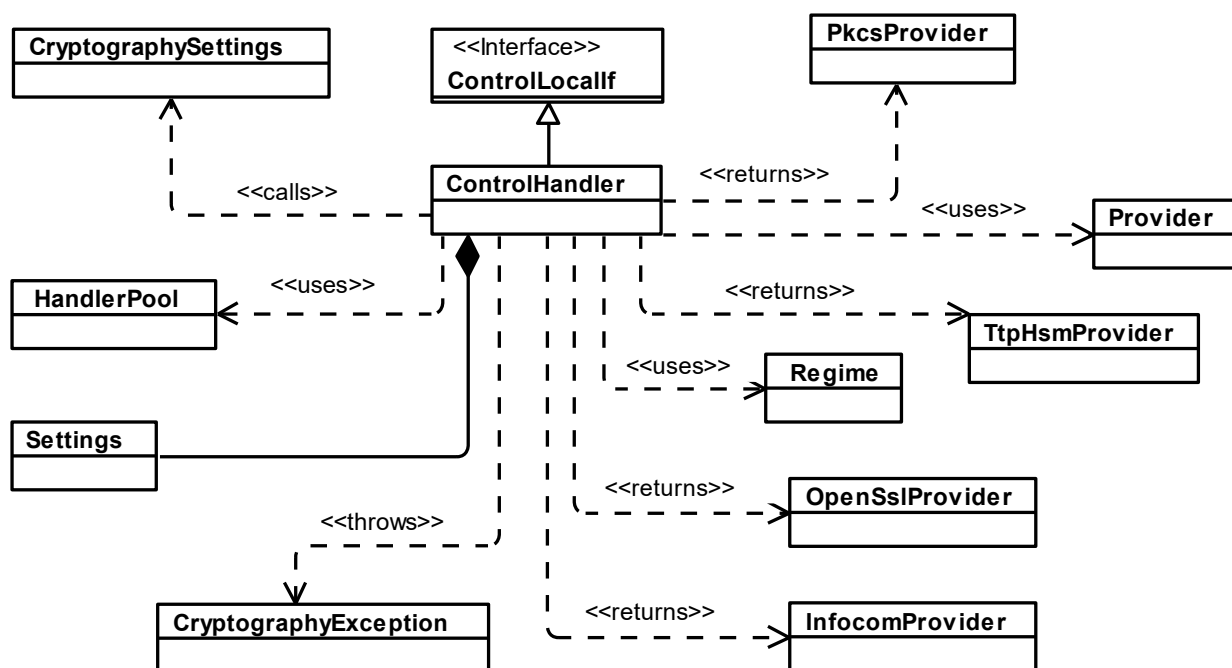


Рисунок 12 – Диаграмма классов обработчика управляющих операций

ControlHandler является обработчиком сервиса управляющих операций. Он реализует интерфейс ControlIf, предоставляя методы управления сервисом криптографии. Основная ответственность заключается в применении новых настроек к системе, а также предоставлении информации о состоянии сервиса. Дополнительно, обработчик управляет множеством доступных обработчиков интерфейса криптографических операций. Управление заключается в перенастройке объекта класса HandlerPool через использование структур, объявленных в интерфейсе: Settings, Provider и Regime. Помимо этого, любой вызов управляющей процедуры в случае неудачи выбрасывает исключение средствами класса CryptographyException.

3.7 Средства реализации

В результате процесса анализа требований к разрабатываемым программным средствам определены основные технологии и средства разработки, необходимые для реализации всех функциональных и нефункциональных требований.

Для написания основного кода проектируемой подсистемы в качестве языка программирования выбран C++ [11]. Такой выбор определен следующими ключевыми факторами:

- объектно-ориентированный дизайн, влияющий на принципы и паттерны архитектуры и проектирования;
- высокая производительность и эффективное использование системных ресурсов;
- возможность взаимодействия с низкоуровневыми структурами данных;
- широкая распространенность языка и развитая экосистема библиотек и инструментов разработки.

Для эффективного управления внешними зависимостями и библиотеками выбран механизм в виде использования специализированного пакетного менеджера. Такой инструмент обеспечивает централизованную установку, обновление и сопровождение сторонних зависимостей, что упрощает этапы интеграции, сборки и сопровождения в процессе реализации программных средств.

В качестве основного и единственного пакетного менеджера выбран Conan. Пакетный менеджер Conan является современным решением для управления внешними зависимостями в проектах на языках программирования C и C++ [12]. Основным механизмом взаимодействия с Conan является составление описания устанавливаемых сторонних зависимостей посредством файла, написанного на языке Python. В таком файле описываются устанавливаемые зависимости, настройки и опции для параметров сборки, способ получения исходного кода, описание процесса сборки и упаковки. Такой выбор определен следующими ключевыми факторами:

- широкая поддержка библиотек C и C++;
- интеграция с системами сборки;
- контроль версий сторонних зависимостей;

- поддержка кроссплатформенности;
- возможность создания собственных пакетов.

Для простой и эффективной возможности собирать реализованные части исходного кода проекта и включать сторонние зависимости в единое целое выбран механизм в виде использования систем сборки.

В качестве системы сборки выбран инструмент CMake. Такой инструмент является стандартом для проектов на C и C++, обеспечивая кроссплатформенную автоматизацию процесса конфигурирования и сборки программного обеспечения [13]. CMake позволяет создавать универсальные и гибко настраиваемые сценарии сборки исходного кода, поддерживающие различные компиляторы и операционные системы. Использование CMake предоставляет следующие преимущества:

- интеграция с пакетным менеджером Conan;
- генерация проектов для различных IDE и платформ;
- гибкость при настройке параметров сборки;
- широкое распространение и качественная документация.

В качестве современной распределенной системы управления версиями, широко используемой в процессе разработки программного обеспечения, принято использовать Git.

Git обеспечивает механизмы отслеживания изменений в исходном коде, организации командной работы, а также возможности слияния различных веток разработки и восстановления предыдущих версий проекта.

Операционной системой, на уровне которой происходит разработка и отладка программных средств, является Debian. Debian можно считать стабильной и надежной операционной системой семейства Linux, которая широко используется в серверных и рабочих средах.

Кроме того, Debian отличается высоким уровнем безопасности, интеграцией с большим количеством пакетов и долгосрочной поддержкой версий.

3.8 Сторонние зависимости

С целью обеспечения функциональности по предоставлению конфигурируемости, логирования, сериализации данных и тестирования в проекте используются сторонние зависимости.

Библиотека `fmt` используется для возможности форматировать строки в стиле Python. С ее помощью становится доступно формирование читаемых сообщений, построения журналов и форматирования текстовых представлений. Является крайне производительным и безопасным способом работы со строками.

Высокопроизводительный модуль логирования `spdlog` применяется для записи событий, ошибок и диагностической информации. Совместно с `fmt` обеспечивает форматируемый вывод сообщений и поддерживает различные уровни логирования.

Современная и удобная работа с JSON поставляется `nlohmann::json`. Библиотека используется для загрузки и сериализации конфигурационных файлов, обмена данными между компонентами, а также для преобразования структур, описанных в интерфейсах Apache Thrift.

Модульное тестирование разрабатываемых компонент становится доступно средствами фреймворка `GoogleTest`. Фреймворк применяется для проверки корректности реализации компонентов разрабатываемой подсистемы инструментами тестовых фикстур, параметризованных тестов и отчетов о результатах выполнения.

Управление установкой и версионированием описанных зависимостей осуществляется с помощью пакетного менеджера `Copan`, что упрощает процесс сборки и обеспечивает воспроизводимость окружения.

4 Реализация подсистемы унифицированного интерфейса

4.1 Интерфейс процедур удаленного вызова

После составления описаний интерфейсов следующим этапом в процессе реализации является выполнение генерации исходного кода на требуемом языке программирования. Такая процедура осуществляется с помощью специального компилятора Thrift, который принимает в качестве входных данных файлы с расширением .thrift и генерирует набор классов и вспомогательных функций на выбранном языке программирования, где в данном случае – на C++. Сгенерированный код содержит каркас для клиентской и серверной частей приложения, что позволяет упростить реализацию логики для обработчиков процедур и клиентских вызовов.

Для наглядности результатов работы Thrift-компилятора составлены диаграммы классов, приведенные на рисунках из приложения Г, на которых показана структура и взаимосвязь сгенерированных классов и интерфейсов [14]. Такие диаграммы позволяют проиллюстрировать, какие классы доступны разработчику после этапа генерации и как они связаны друг с другом.

Сгенерированный Thrift-компилятором код включает классы и компоненты, описание для которых составлено в таблицах 6 и 7.

Таблица 6 – Основные сгенерированные классы интерфейса сервиса криптографических операций

№	Наименование	Описание
1	CryptographyException	Исключение, используемое сервисом для передачи сообщений об ошибках криптографических операций
2	Algorithm	Представление алгоритма
3	ProfileStatus	Статус профиля
4	Status	Статус службы
5	SignRequest	Входные параметры функции Sign
6	SignResponse	Выходные параметры функции Sign
7	VerifyRequest	Входные параметры функции Verify
8	VerifyResponse	Выходные параметры функции Verify
9	DigestRequest	Входные параметры функции Digest
10	DigestResponse	Выходные параметры функции Digest
11	GetSignAlgorithmRequest	Входные параметры функции GetSignAlgorithm

Продолжение таблицы 6

№	Наименование	Описание
12	GetSignAlgorithmResponse	Выходные параметры функции GetSignAlgorithm
13	GetDigestAlgorithmRequest	Входные параметры функции GetDigestAlgorithm
14	GetDigestAlgorithmResponse	Выходные параметры функции GetDigestAlgorithm
15	GetCertificateRequest	Входные параметры функции GetCertificate
16	GetCertificateResponse	Выходные параметры функции GetCertificate
17	GetCryptographyStatusResponse	Выходные параметры функции GetStatus
18	CryptographyIf	Абстрактный интерфейс с методами сервиса криптографии
19	CryptographyIfFactory	Фабрика для создания обработчиков CryptographyIf
20	CryptographyIfSingletonFactory	Фабрика, всегда возвращающая единственный экземпляр CryptographyIf
21	CryptographyNull	Пустая реализация CryptographyIf, не выполняющая операций

Кроме этого, для каждого из методов генерируемого интерфейса составляются следующие структуры:

- Cryptography_<Метод>_args – внутренняя структура для передачи аргументов сервера при вызове метода;
- Cryptography_<Метод>_pargs – внутренняя структура с указателями на аргументы для отправки вызова метода;
- Cryptography_<Метод>_result – внутренняя структура для возвращения результата вызова метода;
- Cryptography_<Метод>_presult – внутренняя структура с указателями на возвращаемые данные при вызове метода.

Таблица 7 – Основные сгенерированные классы интерфейса сервиса служебных операций

№	Наименование	Описание
1	OpenSslProfile	Набор параметров для режима работы через OpenSSL
2	CryptoApiProfile	Набор параметров для режима работы через CryptoApi
3	PkcsProfile	Набор параметров для режима работы через PKCS#11
4	RegimeProfile	Объединение наборов параметров для режимов работы
5	Profile	Профиль использования поставщика криптографии
6	OpenSsl	Режим работы через OpenSsl
7	CryptoApi	Режим работы через CryptoApi
8	TtpHsm	Режим работы через ЭЦП ДТС ЕАЭС
9	Pkcs	Режим работы через PKCS#11
10	Regime	Объединение режимов работы
11	Provider	Поставщик криптографических операций

Продолжение таблицы 7

№	Наименование	Описание
12	Settings	Настройки службы криптографии
13	RemoteSettings	Настройки подключения к удаленной службе
14	Device	Устройство подписи
15	SetSettingsRequest	Входные параметры функции SetSettings
16	SetRemoteSettingsRequest	Входные параметры функции SetRemoteSettings
17	CheckSettingsRequest	Входные параметры функции CheckSettings
18	CheckSettingsResponse	Выходные параметры функции CheckSettings
19	CheckRemoteSettingsRequest	Входные параметры функции CheckRemoteSettings
20	GetSettingsResponse	Выходные параметры функции GetSettings
21	GetRemoteSettingsResponse	Выходные параметры функции GetRemoteSettingsStatus
22	GetProvidersResponse	Выходные параметры функции GetProviders
23	GetDevicesResponse	Выходные параметры функции GetDevices
24	GetAlgorithmsResponse	Выходные параметры функции GetAlgorithms
25	GetControlStatusResponse	Выходные параметры функции GetControlStatus
26	GetRemoteStatusResponse	Выходные параметры функции GetRemoteStatus
27	ControlIf	Абстрактный интерфейс с методами сервиса локального и удаленного управления
28	ControlIfFactory	Фабрика для создания обработчиков ControlIf.
29	ControlIfSingletonFactory	Фабрика, всегда возвращающая единственный экземпляр ControlIf
30	ControlNull	Пустая реализация ControlIf, не выполняющая операций
31	ControlLocalIf	Абстрактный интерфейс с методами сервиса локального управления
32	ControlLocalIfFactory	Фабрика для создания обработчиков ControlLocalIf.
33	ControlLocalIfSingletonFactory	Фабрика, всегда возвращающая единственный экземпляр ControlLocalIf
34	ControlLocalNull	Пустая реализация ControlLocalIf, не выполняющая операций

Аналогично тому, как создавались внутренние структуры для интерфейса криптографических операций, были реализованы внутренние структуры для интерфейса локальных и удаленных операций управления службой криптографии.

4.2 Использование кодогенерации

Использование фреймворка Apache Thrift позволяет создавать кодогенерацию для описанных интерфейсов, что было отмечено ранее. По этой причине необходимо включить сгенерированные файлы в проект средствами инструмента сборки CMake. На рисунке 13 представлена диаграмма

компонентов с точки зрения построения библиотек для произведенной кодогенерации.

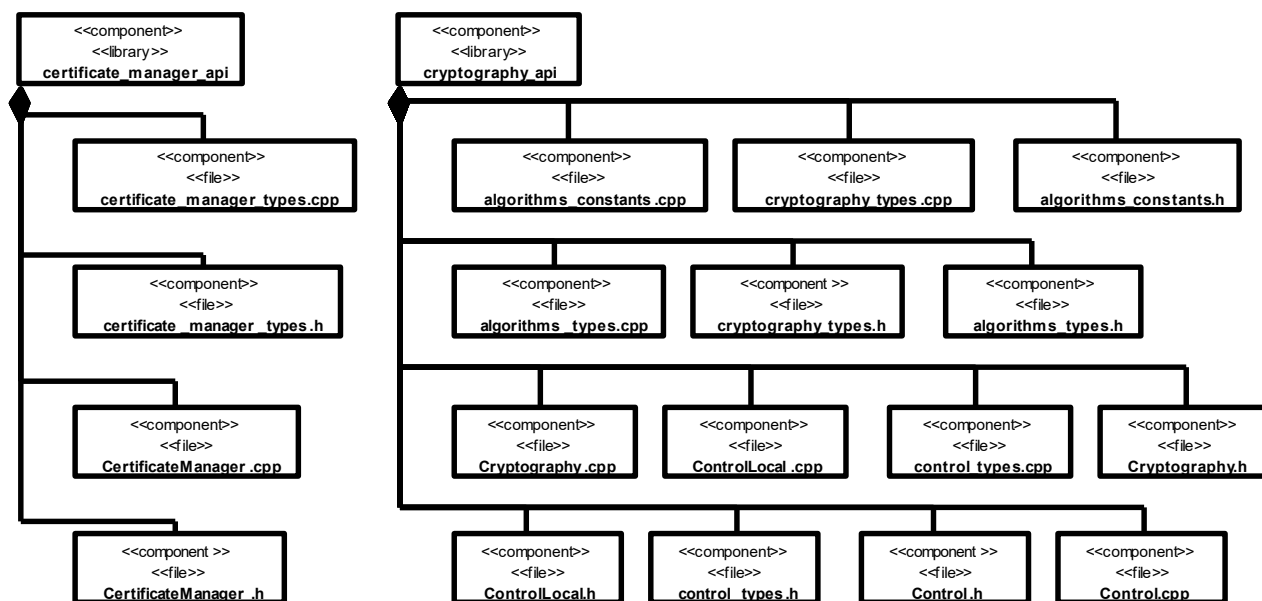


Рисунок 13 – Интеграция сгенерированных файлов в структуру проекта

Включение файлов в проект позволяет получать доступ к коду в этих файлах.

4.3 Слойная схема Apache Thrift

Следующим шагом важно отметить, что точки входа микросервиса должны быть построены на инфраструктуре Apache Thrift, а значит, должны разделять единую слоевую схему инициализации. Слойная схема настройки и инициализации Apache Thrift выглядит следующим образом:

- тип сервера – обеспечивает обработку запросов в рабочих потоках приложения;
- процессор – инкапсулирует маршрутизацию входящих вызовов к соответствующему обработчику;
- обработчик – класс-наследник сгенерированного интерфейса, который содержит бизнес-логику процедуры;

- транспорт – слой буферизации и фреймирования, отвечающий за надежную доставку байтов поверх сокета;
- протокол – способ сериализации структур данных в последовательность байтов;
- сокет – принимает входящие соединения на заданном IP-адресе и порту.

Ниже приведён листинг 1, демонстрирующий запуск Thrift-сервера с единственной точкой входа `CryptographyIf`.

Листинг 1 – Минимальная функция `main` для запуска Thrift-сервера

```
#include "Cryptography.h"

int main() {
    auto handler = std::make_shared<CryptographyHandler>();
    auto processor =
std::make_shared<CryptographyProcessor>(handler);
    auto serverTransport = std::make_shared<TServerSocket>(9090);
    auto transportFactory =
std::make_shared<TBufferedTransportFactory>();
    auto protocolFactory =
std::make_shared<TBinaryProtocolFactory>();

    TThreadedServer server(processor, serverTransport,
transportFactory, protocolFactory);
    server.serve();
}
```

Заданный листинг описывает процесс из создания объектов для каждого уровня слоевой схемы настройки Apache Thrift. Множество составленных объектов далее применяется при создании объекта класса сервера, который после применяет метод, производящий обработку входящих и исходящих вызовов. Исполнение программы не завершается до тех пор, пока сервер не прекратит собственное выполнение.

4.4 Реализация бизнес-логики и обработчиков

Учитывая заданное описание слоевой схемы настройки Apache Thrift, становится ясно, что основной задачей при реализации микросервиса является реализация слоя обработчиков. По этой причине для обработки удаленных

вызовов процедур интерфейса криптографии необходимо составить наследуемый класс, который будет реализовывать все заданные интерфейсом процедуры.

Исходя из требований в оптимальном взаимодействии с динамическим модулем PKCS#11 и его однократной инициализацией, созданием объекта сессии, поиска ключевой пары и определения типов применяемых механизмов составлен класс PkcsHandlerFactory. Класс PkcsHandlerFactory составлен с применением паттерна «Фабрика» для достижения создания обработчиков вызовов относительно ранее проведенной инициализации. Чтобы гарантировать сохранность всей информации, полученной из процесса инициализации, составлен класс ActiveProfile, представляющий профиль, для которого выполнена «активация».

Для выполнения построения бизнес-логики микросервиса в качестве обработчика вызовов интерфейса сервиса криптографии составлен класс PkcsHandler. Указанный класс применяет наследование от сгенерированного интерфейса CryptographyIf, что дает доступ на использование класса в инфраструктуре Apache Thrift. В качестве параметра при конструировании экземпляра класса используется объект класса ActiveProfile, который составляется ранее описанным PkcsHandlerFactory при инициализации динамического модуля. Основное взаимодействие экземпляра PkcsHandler задается применением объекта сессии в методах выполнения криптографической подписи и хеширования.

Операция проверки подписи класса PkcsHandler из-за необходимости в поддержке криптографических алгоритмов не только стандарта ГОСТ, но также СТБ и RSA, выполнена в виде отдельных структур, использующих в наследовании интерфейс SignatureVerifier, объявляющий следующий метод:

– Verify – выполнение верификации подписи относительно подписанных данных, подписи и информации об открытом ключе в виде структуры данных из сертификата.

Основной логикой в методе верификации является определение типа открытого ключа, параметров механизма подписи, составление шаблона открытого ключа для его импорта на устройство подписи, определение механизма верификации и применение объекта авторизованной сессии для выполнения самой верификации средствами загруженного динамического модуля. В общей сложности реализовано три класса относительно каждого стандарта, что помечено в их названии: `GostVerifier`, `StbVerifier` и `RsaVerifier`.

Из-за необходимости в определении перечня доступных к использованию устройств подписи класс `PkcsHandlerFactory` вместе с определением логики создания обработчиков вызовов наследуется от интерфейса `HandlerFactoryIf` со следующими методами:

- `GetHandler` – позволяет создать обработчик вызовов;
- `GetDeviceGatherer` – позволяет создать сборщик устройств подписи.

В интересах следования принципам SOLID внутренняя логика по обнаружению подключённых и поддерживаемых устройств подписи вынесена в класс `PkcsDeviceGatherer`. Указанный компонент наследуется от интерфейса `DeviceGathererIf`, который объявляет лишь одну функцию:

- `Gather` – позволяет обнаружить и составить коллекцию из разрешенных устройств подписи.

Определение метода `Gather` в классе `PkcsDeviceGatherer` выполняет взаимодействие с динамически загруженным модулем PKCS#11 для получения требуемого списка, после чего производит доработку во внешние структуры и передает определенное множество на верхние уровни стека вызовов.

Обработчик вызовов интерфейса операций управления и настройки сервиса криптографии получил свое воплощение в классе `ControlHandler`. Класс реализован в потокобезопасном стиле за счет использования объекта мьютекса. Основной выполняемой задачей класса является применение структуры настроек, передаваемой по RPC интерфейсу. Относительно этой задачи происходит дополнение недостающих частей переданных настроек, их

валидация, определение поставщика криптографических операций, режима работы поставщика криптографических операций, обновление множества обработчиков вызовов и сохранение переданных настроек. Вместе с методом применения настроек класс реализует метод для проверки настроек без их применения и множество методов для получения информации о текущем состоянии интерфейса криптографических операций.

С учетом необходимости соблюдения требования по выполнению протоколирования произведено создание логирующего объекта в виде функции, реализующей паттерн синглтона Майерса. Внутри функции создается статический экземпляр класса логирования `Logger` и выполняется его инициализация. Применение такой реализации позволяет гарантировать ленивую инициализацию объекта, потокобезопасность относительно доступа к объекту и обеспечить единый доступ к объекту протоколирования на всем протяжении выполнения программы.

Необходимость в определении константы, нотирующей криптографический механизм, относительно OID алгоритма удовлетворена в виде константных ассоциативных неупорядоченных массивов, где представлен перечень доступных для использования криптографических констант. Доступ к заданным массивам составлен в виде классов `SignMechanism` и `DigestMechanism`, которые выполняют алгоритма и типа механизма относительно переданного OID в собственном конструкторе. Представленные классы реализованы с использованием полиморфизма времени компиляции, паттерн для которого обозначается CRTP (Curiously Recurring Template Pattern).

Управление исключениями и гарантия корректности при выполнении операций с библиотеками PKCS#11 реализована в классе `PkcsException`. Полями класса являются низкоуровневый код ошибки, наименование функции, где произошла проблема, и сообщение об исключении в человекочитаемом виде. Составление соотнесения численного кода ошибки и сообщения, которое он значит, представлено в специальной функции `ReturnValue` с использованием языковой конструкции `switch`. Любое исключение, проявляемое на стороне

реализовываемых служб, оборачивается в генерируемую структуру `BackupManagerException`, описание для которой представлено в RPC-интерфейсе. Таким образом удастся логироваться и передавать на сторону клиента информацию об успешности выполнения запрашиваемого функционала. Кроме того, такой подход позволяет обеспечить унификацию семантики отказов вне зависимости от используемого производителя токенов.

Наличие в процессе разработки таких библиотек как `nlohmann::json` и `fmt` привело к тому, что для взаимодействия сгенерированных структур, определенных в интерфейсах служб, необходимо составить адаптеры для преобразования структур в представления в виде структур и методов, которые позволят взаимодействовать с упоминаемыми библиотеками. Так, в результате был составлен следующий перечень макросов:

- `NLOHMANN_DEFINE_THRIFT_STRUCT_FIELD_TO` – устанавливает значение поля структуры в объект JSON под ключом с именем поля;
- `NLOHMANN_DEFINE_THRIFT_STRUCT_TO` – определяет неинтрузивную функцию для структуры, обеспечивающую сериализацию всех полей;
- `NLOHMANN_DEFINE_THRIFT_STRUCT_FIELD_FROM` – устанавливает поле структуры из JSON;
- `NLOHMANN_DEFINE_THRIFT_STRUCT_FROM` – определяет неинтрузивную функцию для структуры, обеспечивающую десериализацию всех полей;
- `NLOHMANN_DEFINE_THRIFT_STRUCT_NON_INTRUSIVE` – определяет описания функций сериализации и десериализации для структуры;
- `NLOHMANN_DEFINE_THRIFT_UNION_FIELD_TO` – проверяет наличие поля в объединении и, если поле существует, сериализует его в JSON;

- `NLOHMANN_DEFINE_THRIFT_UNION_TO` – определяет неинтрузивную функцию для объединения, обеспечивающую сериализацию всех вариантов;
- `NLOHMANN_DEFINE_THRIFT_UNION_FIELD_FROM` – для объединения проверяет наличие поля в JSON и задаёт соответствующее значение;
- `NLOHMANN_DEFINE_THRIFT_UNION_FROM` – определяет неинтрузивную функцию для объединения, обеспечивающую десериализацию всех вариантов;
- `NLOHMANN_DEFINE_THRIFT_UNION_NON_INTRUSIVE` – определяет описания функций сериализации и десериализации для объединения.

Вместе с тем, как был составлен перечень макросов, определено множество перегрузок для класса `fmt::formatter`, которые позволяют выполнять использование сгенерированных структур в строках с семантикой форматирования, определенной библиотекой `fmt`.

Диаграмма для классов, описанных в разделе, представлена на рисунке Д.1.

4.5 Маршрутизация и управление обработчиками

Возможность маршрутизации вызовов между удаленными и локальными сервисами интерфейса криптографических операций определена в виде класса `CryptographyRouter`, который все так же наследуется от сгенерированного интерфейса. Основной выполняемой задачей класса является выполнение маршрутизации, логика для которой определена в специальном методе по имени `Route`. Логика метода сосредоточена на условном ветвлении, условием в котором является наличие объекта, указывающего на удаленное взаимодействие. Относительно первой или второй условной ветви возвращается специализированный объект, обрамляющий идеи о кешировании и применении паттерна RAII.

В виде, который похож на реализацию класса `CryptographyRouter`, построен класс `ControlRouter`, который выполняет смежную задачу по маршрутизации вызовов между удаленными и локальными сервисами, но только для интерфейса операций управления сервисом криптографии. Отличительной особенностью в `ControlRouter` относительно `CryptographyRouter` является наличие механизмов для кеширования клиентских подключений с целью ленивого инстанцирования клиентских подключений и их дальнейшего использования при получении вызовов удаленных процедур.

Для выполнения оптимизации работоспособности микросервиса реализован механизм кеширования обработчиков удаленных вызовов процедур. Механизм кеширования полагается на структуру в виде неупорядоченного ассоциативного массива с возможностью использования повторяющихся ключей. Для такой структуры ключами являются именования профилей, относительно которых был построен обработчик вызовов. Поверх кеширующей структуры составлен класс `HandlerCache`, который определяет специальный интерфейс с однозначными методами:

- `Retrieve` – позволяет получить обработчик относительно имени профиля, если таковой имеется, иначе использовать механизм создания нового обработчика для все того же профиля;
- `Return` – позволяет вернуть обработчик в кеширующую структуру.

Относительно класса `HandlerCache` составлен класс `HandlerCacheConcurrent`, который реализует все тот же интерфейс, но позволяет производить работу с кеширующей структурой в многопользовательском режиме. Конкурентное взаимодействие реализовано за счет использования механизмов контекстной блокировки мьютекса с управлением времени жизни самой блокировки в виде объекта через использование `RAII` (`Resource Acquisition Is Initialization`). Контекстные блокировки применены в двух видах: с монопольным владением, при котором только один поток может работать с

защищённым ресурсом, и с разделяемым доступом, допускающим одновременное использование ресурса несколькими потоками в режиме чтения.

Для реализации паттерна RAII при взаимодействии с классами, управляющими кеширующей структурой, реализован класс `HandlerLease`, который позволяет для конкретного экземпляра и именованного профиля произвести получение обработчика при конструкции объекта, а его возврат, соответственно, при уничтожении [15].

С целью поддержки работы со множеством обработчиков вызовов одновременно реализован класс `HandlerPool`, который определяет идею о поддержке пула обработчиков, а также наследуется от интерфейса криптографических операций, что делает его доступным для использования в части слоев инфраструктуры Apache Thrift. В основе своей этот класс полагается на ранее определенные `HandlerLease` и `HandlerCacheConcurrent`.

Диаграммы для классов, описанных в разделе, представлены на рисунках Д.2 и Д.3.

4.6 Настройка и конфигурация микросервиса

Удовлетворение о конфигурируемости исполняемого файла сервиса криптографии отражено при построении классов `CryptographySettings`, `PkcsProvider` и `Endpoint`. Представленный класс `CryptographySettings` позволяет произвести настройку адреса для точки доступа к управлению службой криптографии, точки доступа к целевым криптографическим функциям, а также точки доступа для диагностики состояния. Вместе с настройкой точек доступа представлена возможность для определения перечня доступных к использованию поставщиков криптографических операций, множества доступных алгоритмов подписи и хеширования, а также путь для файла, который определяет текущий заданный уровень для протоколирования. Инициализация структуры настроек происходит в момент запуска исполняемого файла и происходит в виде чтения и сериализации данных из JSON файла.

Использование в качестве основной операционной системы Debian 11 позволяет применять обширную инфраструктуру systemd в процессе разработки, отладки и последующем распространении программного продукта [16]. Наличие такой возможности привело к составлению файла `cryptography.service`, определяющего разнообразные параметры для регистрации исполняемого файла в качестве systemd-юнита. Данный файл описывает тип сервиса, способ его перезапуска и интервал перезапуска. Также задаются пути к директориям конфигурации и файлам, таким как конфигурационный файл и файл с уровнем логирования.

Важно отметить использование переменных окружения, что позволяет настраивать работу сервиса без изменения исходного кода. Кроме того, файл определяет каталоги времени исполнения и состояния, логирование событий в журнал systemd и устанавливает необходимые зависимости и цели при запуске через опцию `[Install]`. Такое детальное конфигурирование упрощает эксплуатацию микросервиса и повышает его надёжность при различных сценариях использования.

Для корректного завершения работы со стороны разрабатываемого микросервиса реализован паттерн Graceful Shutdown [17]. Реализация паттерна строится на обработке принимаемых сигналов со стороны операционной системы.

Так, при получении сигнала `SIGINT` или `SIGTERM` микросервис прекращает прием новых соединений, дожидается завершения активных запросов, производит очистку динамически загруженной информации, сбрасывает сессии токенов и пишет контрольную запись в журнал аудита. Это предотвращает потерю данных и неконсистентные состояния как для инфраструктуры, предоставляемой PKCS#11, так и для операционной системы, в рамках которой ведется исполнение.

Диаграмма для классов, описанных в разделе, представлена на рисунке Д.4.

4.7 Вспомогательные утилиты

Необходимость со стороны команды разработчиков в применении и выполнении запросов RPC-интерфейса привела к проектированию и построению вспомогательной утилиты для выполнения клиентских вызовов. Выполнение утилиты располагается в основном исполняемом файле и строится на обработке передаваемых при запуске строковых параметров (листинг 2).

Листинг 2 – Варианты применения клиентской утилиты для взаимодействия с интерфейсами сервиса

```
# Операции без указания типа используемого сервиса и параметра
./cryptography --help
./cryptography --get-sign-algorithm

# Операции с указанием типа используемого сервиса
./cryptography resign --digest
./cryptography worker.cross --sign

# Операции с указанием типа используемого сервиса и параметра
./cryptography tsp.national --set-settings ./set-settings
./cryptography develop --verify {"data": "...", "pki": "...",
"profile": "...", ...}}
```

Содержание листинга 2 среди вариантов показывает способ применения утилиты при обращении к сервису с названием `tsp.national` для установки параметров работы сервиса криптографических операций. Как можно заметить, передача параметров для операции удаленного вызова ведется через предоставление пути до файла в формате JSON, который содержит в себе сериализованную структуру из описания интерфейса. Такой подход позволил упростить использование и сэкономить количество передаваемых параметров в командной строке, что являлось главным пожеланием при построении.

Вместе с необходимостью в утилите для выполнения клиентских вызовов возникла потребность в проведении стресс-тестирования для проверки качества работоспособности разрабатываемых программных средств. Аналогично утилите клиентских вызовов, выполнение утилиты для стресс-тестирования располагается в основном исполняемом файле и строится на обработке проверки,

что первый передаваемый при запуске приложения параметр соответствует строке stress (листинг 3).

Листинг 3 – Применение утилиты для проведения стресс-тестирования

```
./cryptography stress --help
./cryptography stress develop --set-settings
./cryptography stress ./setup resign --get-status
./cryptography stress resign --set-remote-settings ./set-remote-
settings
./cryptography stress ./setup worker.cross --set-settings ./set-
settings
```

Важным отличием в подходе при выполнении стресс-тестирования является возможность передача пути до файла с параметрами выполнения тестирования. Среди таких параметров есть возможность указать количество потоков, которые выполняют запросы, а также количество запросов, которые исполняются в одном потоке.

Результат выполнения стресс-тестирования формируется в виде сообщения, которое далее выводится на стандартный поток. Пример того, как может выглядеть такое сообщение, представлен на листинге 4.

Листинг 4 – Пример результата стресс-тестирования

```
----- Результаты стресс-тестирования -----
| Общее время | 12 345 ms |
| Запросов всего | 50 000 |
| Неудачных запросов | 42 |
| Вероятность ошибки | 0,08% |
| Пропускная способность | 4,05 req/ms |
| Задержка P50 / P90 / P99 | 50,2/ 120,7/ 350,9 |
-----
```

Изложенные архитектурные и программные решения демонстрируют ключевые принципы работы микросервиса: от инициализации слоёв инфраструктуры Apache Thrift и управления жизненным циклом обработчиков до обеспечения надёжности через обработку исключений и корректное завершение работы. Представленные примеры кода иллюстрируют базовый механизм запуска и взаимодействия компонентов, а вспомогательные утилиты – способы управления и тестирования сервиса. Диаграмма для классов, описанных в разделе, представлена на рисунке Д.5.

5 Контроль стандартов качества программных решений

5.1 Функциональное тестирование

Функциональное тестирование разработанной подсистемы с унифицированным интерфейсом на основе протокола PKCS#11 проведено с целью проверки работоспособности основных реализованных функций и подтверждения их корректного выполнения в соответствии с предъявленными требованиями. В рамках данного этапа были проверены следующие возможности системы:

- операции электронной цифровой подписи;
- операции криптографического хеширования данных;
- операции проверки электронной подписи;
- функции настройки подсистемы;
- функции управления подсистемой.

Проверка осуществлялась на стороне приложения, предоставляющего унифицированный PKCS#11-интерфейс. Это означало, что все тестовые обращения к криптографическим операциям выполнялись через разработанный интерфейс приложения, который взаимодействует с различными криптографическими модулями. Такой подход позволил убедиться, что интеграция подсистемы с внешними криптографическими средствами работает корректно, а само приложение правильно обрабатывает запросы и возвращает ожидаемые результаты. Тестирование охватывало как типичные сценарии использования (нормативные случаи), так и граничные условия и нештатные ситуации, когда входные данные содержат ошибки или выходят за допустимые пределы.

Для каждой категории функций были подготовлены наборы входных данных и ожидаемых результатов. Криптографические операции проверялись путем сравнения результатов работы подсистемы с эталонными значениями или известными корректными примерами. В частности, операция подписи

тестировалась путем формирования цифровой подписи на заданном сообщении с последующей успешной проверкой этой подписи с использованием соответствующего открытого ключа. Операция хеширования проверялась сопоставлением вычисленного хеш-значения с предварительно вычисленным эталоном. Для функции проверки подписи рассматривались как случаи правильной верификации, так и случаи намеренного искажения данных или подписи, чтобы удостовериться, что система выдаёт сообщение об ошибке при несоответствии. Функции настройки и управления подсистемой тестировались посредством вызовов соответствующих методов удаленного интерфейса с различными параметрами конфигурации. Например, проверялась возможность задать новые настройки криптографического провайдера через приложение: передавались корректные и некорректные конфигурационные данные, после чего анализировалась реакция системы. В штатных случаях сервис должен был принять и применить переданные настройки, подтверждая успех операции, а при обнаружении ошибок в параметрах – возвращать информативные сообщения об ошибке. Аналогично тестировались управляющие функции, связанные с контролем работы сервиса, с проверкой того, что каждая команда выполняется должным образом и приводит систему в ожидаемое состояние.

Результаты функциональных тестов зафиксированы и сведены в рисунки, представленные в приложении Е. Эти результаты наглядно демонстрируют выходные данные и реакции подсистемы на различные сценарии тестирования для каждой из перечисленных функций.

Анализ полученных данных показал, что во всех рассмотренных тестовых сценариях подсистема ведет себя корректно: криптографические операции возвращают ожидаемые значения, функции конфигурирования успешно применяют заданные настройки, а при некорректных входных данных система выдаёт предусмотренные сообщения об ошибках.

5.2 Функциональное тестирование средствами причинно-следственного графа и попарного тестирования

Пользователь должен иметь возможность задать настройки, используемые сервисом, средствами RPC интерфейса. В настройках сервиса должна передаваться информация об используемом поставщике криптографических функций, набор профилей, определяющих параметры работы, используемые поставщиком функций, а также временная метка, которая указывает на дату создания структуры настроек и задается в формате Unix time. Структура с описанием поставщика криптографических функций содержит именование поставщика, его описание и режим работы. Набор профилей задается парой ключ-значение, где ключ – имя, а значение – структура профиля. Структура профиля содержит информацию об алгоритме хеширования, бинарные данные сертификата и его поля в разобранном виде, цепочку сертификатов и специфичные для режима работы параметры. Алгоритм хеширования задается структурой из строки в виде OID, указывающей на алгоритм, строки в виде URI, которая также указывает на алгоритм, строки в виде OID, указывающей на параметры, используемые с алгоритмом, а также описание алгоритма. Специфичные для режима работы PKCS параметры содержат идентификатор используемого устройства подписи и пароль, аутентифицирующий на этом устройстве.

Пользовательский ввод определяет множество реакций, возвращаемых программными средствами, в которые входят:

- сообщение о неверном значении временной метки;
- сообщение о неверном именовании поставщика криптографических операций или отсутствии поддержки запрашиваемых операций и алгоритмов;
- сообщение о неверной структуре режима работы поставщика или несоответствии ожидаемой;
- сообщение о неверных идентификаторах для алгоритма хеширования или его параметров;

- сообщение об отсутствии устройства с заданным идентификатором или неверном аутентифицирующем пароле;
- сообщение об успешном применении переданных настроек.

Множество реакций на пользовательский ввод задает набор причин и следствий, представленный на таблице 8.

Таблица 8 – Набор причин и следствий

Причины	
П1	Временная метка является числом и входит в допустимые пределы
П2	Наименование поставщика криптографических операций указано корректно, и поставщик поддерживает нужные операции и алгоритмы
П3	Структура, определяющая режим работы поставщика, передана корректно и соответствует ожиданиям
П4	OID и URI для алгоритма хеширования, а также OID параметров указаны корректно и соответствуют поддерживаемым алгоритмам
П5	Устройство подписи с заданным идентификатором существует и пароль для аутентификации указан корректно
Следствия	
С1	Сообщение: «Временная метка не является числом или не входит в допустимые пределы»
С2	Сообщение: «Наименование поставщика криптографических операций указано некорректно или поставщик не поддерживает нужные операции и алгоритмы»
С3	Сообщение: «Передана некорректная структура режима работы поставщика или не соответствует ожидаемой»
С4	Сообщение: «OID и URI для алгоритма хеширования, а также OID параметров указаны некорректно или не соответствуют поддерживаемым алгоритмам»
С5	Сообщение: «Устройство подписи с заданным идентификатором отсутствует или пароль для аутентификации указан некорректно»
С6	Сообщение: «Настройки успешно заданы»

Набор причин и следствий определяет функциональную диаграмму, представленную на рисунке 14.

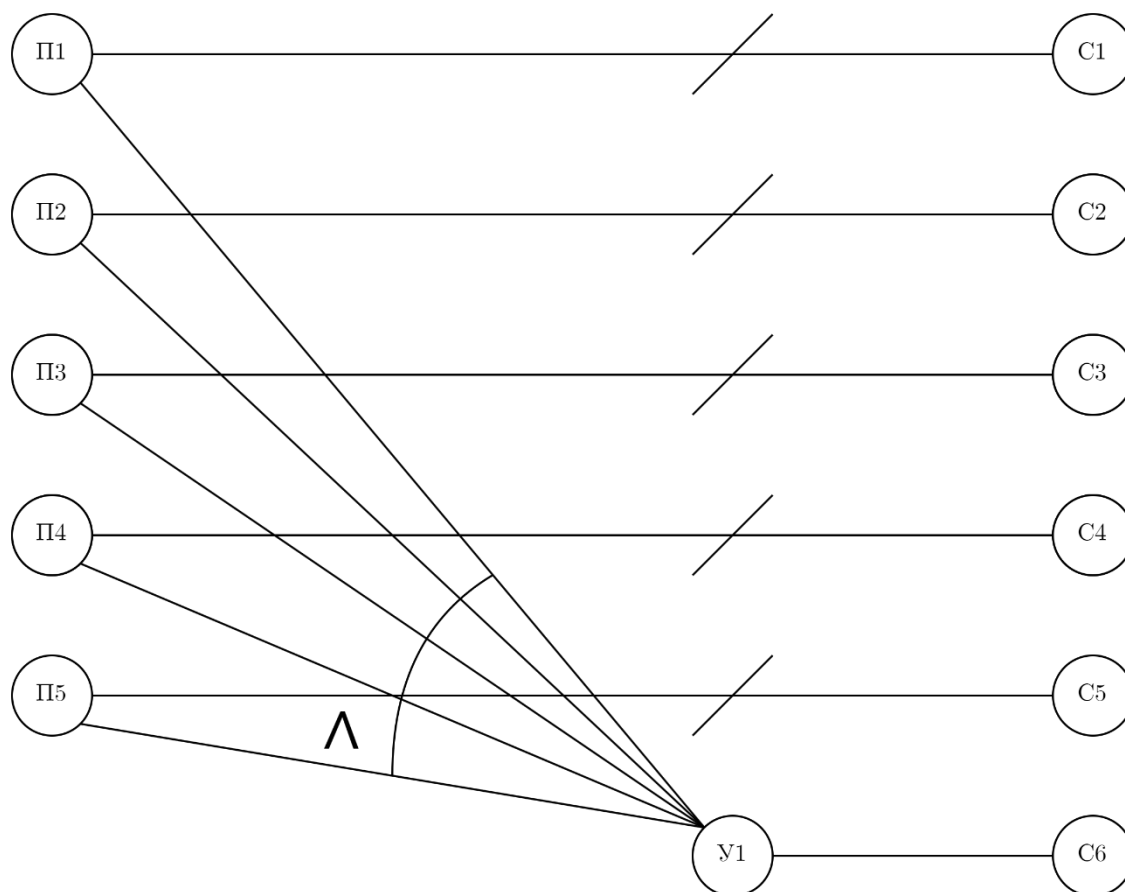


Рисунок 14 – Функциональная диаграмма

Для множества причин, используемых в таблице решений, составлены таблицы 9 - 13 с представлением допустимых ограничений.

Таблица 9 – Перечень ограничений для причины 1

Является числом	Входит в допустимые пределы
Да	$0 - 2^{64} - 1$
Нет	< 0
	$> 2^{64} - 1$

Таблица 10 – Перечень ограничений для причины 2

Наименование поставщика криптографии корректно	Поставщик криптографии поддерживает необходимые операции и алгоритмы
Да	Да
Нет	Нет

Таблица 11 – Перечень ограничений для причины 3

Структура режима работы поставщика корректна	Структура режима работы поставщика соответствует ожидаемой
Да	Да
Нет	Нет

Таблица 12 – Перечень ограничений для причины 4

OID алгоритма хеширования корректен	URI алгоритма хеширования корректен	OID параметров алгоритма хеширования корректен
Да	Да	Да
Нет	Нет	Нет

Таблица 13 – Перечень ограничений для причины 5

Устройство существует	Пароль корректен
Да	Да
Нет	Нет

Таблицы с представлением допустимых ограничений в случае полного перебора всех комбинаций определяют 3072 тестовых сценария. С целью сокращения количества вхождений во множество допустимых тестов используется комбинаторный метод тестирования программного обеспечения, называемый попарным тестированием, который для каждой пары входных параметров системы проверяет все возможные дискретные комбинации этих параметров [18]. Данная техника позволяет уменьшить количество тестовых сценариев с минимальным вмешательством в качество проводимого тестирования.

Результат применения техники попарного тестирования оставил из 3072 тестовых сценариев лишь 10. Тестовые сценарии после применения техники попарного тестирования представлены на таблице 14.

Результат выполнения каждого тестового сценария, определенного на таблице 14, привел к следствию с сообщением «Настройки успешно применены». Таким образом, можно сделать вывод о том, что результат

выполнения технического задания произведен успешно и разработанное программное обеспечение работает корректно.

5.3 Модульное тестирование

С целью обеспечить корректную работоспособность разработанного приложения и определить проверку ожидаемого поведения программы использована техника модульного тестирования. Модульное тестирование представляет собой форму тестирования программного обеспечения, при которой изолированный исходный код тестируется для проверки ожидаемого поведения. Сценарии тестов выполняются на уровне модуля в отличие от тестирования на уровне интеграции или системы.

Тестирование функциональных возможностей реализованного сервиса, разработка которого выполнена на языке C++, исполнено средствами стороннего инструмента тестирования GoogleTest [19, 20]. GoogleTest, также называемый gtest, является средством в виде специализированной библиотеки, используемой для проведения модульного тестирования.

Проверке подверглись ключевые функции подсистемы, включая выполнение криптографических операций, обработку входных параметров и генерацию корректных выходных данных. Результаты тестирования подтвердили соответствие поведения программных компонентов заданным требованиям, что свидетельствует о корректной реализации логики и готовности подсистемы к последующим этапам интеграции.

Выполнение модульного тестирования также позволило сократить время на отладку, поскольку выявление ошибок происходило на ранних стадиях разработки. Это способствовало поддержанию высокого уровня стабильности кода и повышению общей надёжности создаваемого программного обеспечения.

Таким образом, модульные тесты стали неотъемлемой частью процесса верификации корректности реализованной функциональности. Их регулярное выполнение способствует своевременному выявлению регрессий при внесении изменений в код.

Таблица 14 – Тестовые сценарии после применения техники попарного тестирования

№ п/п	1	2	3	4	5	6	7	8	9	10
Является числом	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	Нет
Входит в допустимые пределы	$> 2^{64} - 1$	$0 - 2^{64} - 1$	< 0	$> 2^{64} - 1$	< 0	$> 2^{64} - 1$	$0 - 2^{64} - 1$	< 0	$> 2^{64} - 1$	$0 - 2^{64} - 1$
Наименование поставщика криптографии корректно	Да	Нет	Да	Нет	Да	Нет	Да	Нет	Да	Нет
Поставщик криптографии поддерживает необходимые операции и алгоритмы	Да	Нет	Да	Нет	Нет	Да	Нет	Да	Нет	Да
Структура режима работы поставщика корректна	Да	Нет	Да	Нет	Да	Нет	Да	Нет	Да	Нет
Структура режима работы поставщика соответствует ожидаемой	Да	Нет	Да	Нет	Нет	Да	Нет	Да	Нет	Да
OID алгоритма хеширования корректен	Да	Нет	Да	Нет	Да	Нет	Да	Нет	Да	Нет
URI алгоритма хеширования корректен	Да	Нет	Да	Нет	Нет	Да	Нет	Да	Нет	Да
OID параметров алгоритма хеширования корректен	Да	Нет	Да	Нет	Да	Нет	Да	Нет	Да	Нет
Устройство существует	Да	Нет	Да	Нет	Нет	Да	Нет	Да	Нет	Да
Пароль корректен	Да	Нет	Да	Нет	Да	Нет	Да	Нет	Да	Нет

Множество разработанных модульных тестов представлено на таблице 15. Для каждого из модульных тестов код представлен в приложениях В.3 и В.4.

Таблица 15 – Перечень разработанных модульных тестов

№ п/п	Компонент для тестирования	Описание тестируемого сценария	Ожидаемый результат	Наблюдаемый результат
1	Endpoint	Тестирование нормализации при пустом пути	Путь устанавливается из конфигурации	Объект содержит ожидаемый путь
2	Endpoint	Тестирование нормализации при недопустимом высоком порте	Порт заменяется на значение из конфигурации	Объект содержит корректный порт
3	Endpoint	Тестирование нормализации при отрицательном порте	Порт заменяется на значение из конфигурации	Объект содержит корректный порт
4	Endpoint	Тестирование нормализации при пустом IP-адресе	IP заменяется на 127.0.0.1	Объект содержит IP 127.0.0.1
5	Endpoint	Тестирование, что валидный порт не перезаписывается	Порт остается без изменений	Объект содержит исходный порт
6	Endpoint	Тестирование получения строки подключения при наличии пути	Строка подключения содержит путь	Возвращается путь как строка подключения
7	Endpoint	Тестирование получения строки подключения по IP и порту	Строка подключения в формате IP:порт	Возвращается строка IP:порт
8	Endpoint	Тестирование строки подключения по умолчанию	Строка подключения 127.0.0.1:0	Возвращается строка подключения по умолчанию
9	Endpoint	Тестирование строки подключения с IPv6-адресом	Строка подключения в формате IPv6:порт	Возвращается строка IPv6:порт

Продолжение таблицы 15

№ п/п	Компонент для тестирования	Описание тестируемого сценария	Ожидаемый результат	Наблюдаемый результат
10	CryptographySettings	Нормализация заменяет порты вне допустимого диапазона	Порты устанавливаются на значения по умолчанию	Объект содержит порты 3334, 3335, 3336
11	CryptographySettings	Нормализация не изменяет корректные порты	Порты остаются неизменными	Объект содержит исходные значения портов
12	CryptographySettings	Тестирование, что экземпляр настроек является синглтоном	Обе ссылки указывают на один объект	Адреса двух экземпляров совпадают
13	CryptographySettings	Сброс синглтона восстанавливает значения по умолчанию	Порт сбрасывается до 3334	Объект содержит значение порта 3334 после сброса
14	CryptographySettings	Чтение настроек из потока	Чтение настроек из потока	Объект содержит значения, указанные в JSON
15	ControlHandler	Установка настроек сохраняет их и вызывает сброс пула	Метод Reset вызывается и настройки сохраняются	Метод Reset вызван, имя провайдера – pkcs11
16	ControlHandler	Получение устройств делегировается в пул	Возвращаемый список совпадает с тем, что вернул пул	Объект содержит ожидаемые устройства
17	ControlHandler	Получение статуса делегировается в пул и добавляется временная метка	Метод GetStatus вызывается, временная метка равна 0	Объект содержит статус и временную метку, равную 0
18	ControlHandler	Установка настроек с использованием Infocom провайдера	Метод успешно обрабатывает входящие JSON- настройки	Настройки обрабатываются без ошибок

Совокупность наблюдаемых результатов показала удовлетворительное прохождение для каждого тестового сценария.

5.4 Метрики качества программного обеспечения

В рамках сферы разработки программного обеспечения метрики качества – это стандарт измерения степени, в которой система или ее часть обладает некоторым качеством. Возможность определить степень качества программного кода позволяет оценить сложность разработанного программного обеспечения, представить требуемые усилия по реализации или пониманию функции, а также показать вероятность ошибок в разработанной программе. С целью определить ранее описанные степени качества программного обеспечения принято решение использовать метрики SLOC и Холстеда.

5.4.1 Вычисление и определение метрики SLOC

Метрика «Исходные Строки Кода» (SLOC), также известная как «Строки Кода» (LOC), используется для измерения размера компьютерной программы путем подсчета количества строк в тексте исходного кода программы. Кроме того, метрика позволяет и обычно используется для прогнозирования необходимого объема усилий и трудозатрат, которые потребуются на разработку программного обеспечения на заданном языке программирования, а также для анализа производительности программы или поддерживаемости кода после создания программного обеспечения.

Произведем расчет метрики SLOC на примере кода функции из листинга 5.

Листинг 5 – Код функции определения поставщика криптографии

```
static PkcsProvider FindPkcsProvider(const Provider &provider)
{
    using std::ranges::find;

    // TODO: Заменить `pkcs_providers` на обобщенный вариант
    const auto &pkcs_providers {
        CryptographySettings::Instance().pkcs_providers
    };

    // Определение поставщика из перечня доступных по заданному
    имени
    const auto pkcs_provider { find(pkcs_providers, provider,
                                   &PkcsProvider::provider_base)
    };
};
```

```

// Заданное имя поставщика определяет поставщика
if (pkcs_provider != pkcs_providers.end()) return
*pkcs_provider;

throw CryptographyExceptionExtended {
    "Заданный поставщик криптографических операций недоступен",
};
}

```

Расчет метрики SLOC на примере кода функции из листинга 5 представлен на таблице 16.

Таблица 16 – Расчет метрики SLOC для кода из листинга 5

Метрика	Результат
Физические строки кода	20
Исходные строки кода	13
Строки комментариев	3
Пустые строки	4

Результат расчета метрики показывает, что код функции, представленный на листинге 5, не имеет достаточной сложности, его просто анализировать и использовать.

5.4.2 Вычисление и определение метрики Холстеда

Метрика Холстеда является более комплексной мерой определения качества кода, благодаря которой возможно провести количественный анализ сложности программного кода.

Такая метрика качества основана на подсчете и анализе операторов и операндов. В перечень основных метрик Холстеда входят определение словарного запаса, объема кода, предполагаемой длины, его сложности, усилия для написания и понимания кода, времени написания кода и количества доставленных кодом ошибок.

Словарный запас программы рассчитывается по формуле:

$$\eta = \eta_1 + \eta_2,$$

где η_1 – число уникальных операторов программы;

η_2 – число уникальных операндов программы.

Длина кода программы рассчитывается по формуле:

$$N = N_1 + N_2,$$

где N_1 – общее число операторов в программе;

N_2 – общее число операндов в программе.

Предполагаемая длина кода программы рассчитывается по формуле:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2.$$

Объем кода программы рассчитывается по формуле:

$$V = N \log_2 \eta.$$

Сложность кода программы рассчитывается по формуле:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}.$$

Усилие для написания и понимания кода рассчитывается по формуле:

$$E = D * V.$$

Время написания кода рассчитывается по формуле:

$$T = \frac{E}{18}.$$

Количество доставленных кодом ошибок рассчитывается по формуле:

$$B = \frac{V}{3000}.$$

Произведем расчет метрики Холстеда на примере кода из листинга 5.

Подсчет количества операторов и операндов представлен на таблице 17.

Таблица 17 – Подсчет количества операторов и операндов

№ п/п	Операторы	Число оператор ов	Операнды	Число операн дов
1	::	4	provider	1
2	const	3	pkcs_providers	4
3	auto	2	CryptographySettings	1

Продолжение таблицы 17

№ п/п	Операторы	Число оператор ов	Операнды	Число операн дов
4	&	3	Instance	1
5	{	4	pkcs_provider	3
6	()	3	PkcsProvider	1
7	.	2	provider_base	1
8	}	4	end	1
9	;	5	CryptographyExceptionExtended	1
10	,	3	"Заданный поставщик криптографических операций недоступен"	1
11	if	1	std	1
12	!=	1	ranges	1
13	return	1	find	1
14	*	1		
15	throw	1		
16	static	1		
17	FindPkcsProvid er	1		
18	find	1		
19	Provider	1		
20	PkcsProvider	1		
21	using	1		

Расчет метрик Холстеда представлен на таблице 18.

Таблица 18 – Расчет метрик Холстеда

Метрика	Результат
Словарный запас	$21 + 13 = 34$
Длина кода	$44 + 18 = 62$
Предполагаемая длина	$21 \log_2(21) + 13 \log_2(13) = 92.23 + 48.1 = 140.33$
Объем кода	$62 \log_2(34) = 315.42$

Продолжение таблицы 18

Метрика	Результат
Сложность кода	$\frac{21}{2} * \frac{18}{13} = 14.53$
Усилие для написания	$14.53 * 315.42 = 4585.72$
Время написания	$\frac{4585.72}{18} = 254.76$
Количество доставленных ошибок	$\frac{315.42}{3000} = 0.10514$

Результат расчета метрики Холстеда показал, что программный код рассматриваемой функции удовлетворяет требуемым ограничениям на значения метрик качества. Удовлетворение требуемым ограничениям определяет код рассматриваемой программы как успешный вариант реализации.

Заключение

В результате выполнения выпускной квалификационной работы была разработана подсистема обеспечения унифицированного интерфейса для криптографических токенов на основе протокола PKCS#11. Данная подсистема реализована в виде отдельного программного микросервиса, служащего унифицированным RPC-шлюзом для взаимодействия с различными средствами электронной подписи. В процессе разработки проведен анализ предметной области и существующих решений, выявлены проблемы и ограничения текущих подходов, на основании чего предложено и успешно реализовано эффективное техническое решение.

Разработанный микросервис позволяет существенно упростить интеграцию различных криптографических устройств и библиотек посредством единого типизированного интерфейса, представленного через Apache Thrift. Реализованная архитектура предусматривает динамическую загрузку библиотек, централизованное управление жизненным циклом ключей и поддержку отечественных криптографических алгоритмов.

Проведенное функциональное и модульное тестирование подтвердило работоспособность и надежность разработанного решения. Полученные метрики качества кода свидетельствуют о приемлемом уровне сложности и высокой сопровождаемости программного кода. Корректную работу разработанной подсистемы подтверждает ее внедрение в технологическую платформу компании «ИнфоТеКС», что подтверждено актом о внедрении (приложение Ж).

Эффект от использования предлагаемого микросервиса как подхода выражается в значительном сокращении трудоемкости внедрения и обслуживания криптографических средств, уменьшении возможности привязки к поставщику и обеспечении более гибкой и масштабируемой инфраструктуры криптографической защиты.

Достигнутый результат указывает на достижение поставленных целей и их полное выполнение. Разработанный программный продукт может быть

рекомендован для использования в информационных системах, где требуется надежная, удобная и унифицированная работа с криптографическими устройствами подписи и токенами.

Список использованных источников

1. Microsoft. Cryptography, CryptoAPI, and CAPICOM [Электронный ресурс]. URL: <https://learn.microsoft.com/en-us/windows/win32/seccrypto/cryptography--cryptoapi--and-capicom> (дата обращения: 01.06.2025).
2. OASIS. PKCS #11 Cryptographic Token Interface Base Specification. Version 3.1. – 23 июля 2023 г. – 174 с.
3. КристоПро. КристоПро HSM [Электронный ресурс]. URL: <https://www.cryptopro.ru/products/hsm> (дата обращения: 13.06.2025).
4. ViPNet CSP 4.4.4: руководство пользователя. — М.: АО «ИнфоТеКС», 2022. — 223 с.
5. ИнфоТеКС. ViPNet HSM [Электронный ресурс]. URL: <https://infotecs.ru/products/vipnet-hsm-1-0> (дата обращения: 10.06.2025).
6. Рутокен. Рутокен и ViPNet CSP [Электронный ресурс]. URL: <https://dev.rutoken.ru/display/KB/RU1023> (дата обращения: 15.06.2025).
7. ТехноСервис. Криптографические возможности Рутокен [Электронный ресурс]. URL: https://ts21.ru/helpdesk/faq/chasto-zadavaemye-voprosy/kriptograficheskie_vozmozhnosti_rutoken/ (дата обращения: 17.06.2025).
8. Рутокен. Функции PKCS #11, поддерживаемые устройствами Рутокен [Электронный ресурс]. URL: <https://dev.rutoken.ru/pages/viewpage.action?pageId=3178534> (дата обращения: 17.06.2025).
9. Фаулер М. UML. Основы / М. Фаулер, К. Скотт. – СПб. : Символ-Плюс, 2005. – 192 с.
10. Apache Software Foundation. Документация Apache Thrift [Электронный ресурс]. URL: <https://thrift.apache.org/docs> (дата обращения: 21.05.2025).
11. Ширева С.Н. Основы программирования на языке C/C++ / С.Н. Ширева. – М. : РГППУ, 2017. – 147 с.
12. Conan C/C++ Package Manager Documentation [Электронный ресурс]. – URL: <https://docs.conan.io> (дата обращения: 01.06.2025).

- 13.Дубров, Д.В. Система построения проектов CMake: учебник для вузов / Д.В. Дубров. – Москва : Издательство ЮФУ, 2015. – 39 с.
- 14.Мартин Р. С. Гибкая разработка программ на Java и C++ / Р. С. Мартин. – Москва : Диалектика, 2017. – 704 с.
- 15.Гамма Э.; Хелм Р.; Джонсон Р.; Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб. : Питер, 2020. – 448 с. – ISBN 978-5-4461-1595-2.
- 16.Поттеринг Л. systemd for Administrators: цикл статей [Электронный ресурс]. – URL: <https://github.com/shibumi/systemd-for-administrators/blob/master/systemd-handbook.pdf> (дата обращения: 19.05.2025).
- 17.Graceful exit — Wikipedia, the free encyclopedia [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Graceful_exit (дата обращения: 17.05.2025).
- 18.Кент, Б. Экстремальное программирование. Разработка через тестирование / Б. Кент. – СПб. : Питер, 2022. – 224 с. – ISBN 978-5-4461-1439-9.
- 19.Уиттакер, Дж Как тестируют в Google. / Дж Уиттакер, Дж Арбон, Дж Каролло. – СПб. : Питер, 2014. – 320 с. – ISBN 978-5-496-00893-8.
- 20.GoogleTest User's Guide [Электронный ресурс]. – URL: <https://google.github.io/googletest/> (дата обращения: 23.04.2025).

Приложение А
(справочное)
Глоссарий

UML (Unified Modeling Language) – унифицированный язык моделирования, используемый для визуализации, спецификации, конструирования и документирования программных систем.

HSM (Hardware Security Module) – аппаратный модуль безопасности, предназначенный для безопасного хранения и использования криптографических ключей.

CryptoAPI (Cryptographic Application Programming Interface) – криптографический программный интерфейс Windows, предоставляющий приложениям доступ к функциям шифрования и цифровой подписи.

PKCS#11 (Public-Key Cryptography Standards #11) – стандарт интерфейса прикладного программирования для работы с криптографическими токенами и аппаратными модулями безопасности.

Cryptoki – альтернативное название интерфейса PKCS#11, обозначающее криптографический токен-интерфейс.

API (Application Programming Interface) – интерфейс прикладного программирования, предоставляющий программам возможность взаимодействовать с другими компонентами или сервисами.

CNG (Cryptography Next Generation) – криптографический API, разработанный Microsoft в рамках платформы Windows для замены CryptoAPI и поддержки современных алгоритмов.

Java JCA (Java Cryptography Architecture) – архитектура криптографического интерфейса Java, обеспечивающая доступ к шифрованию, хэшированию и управлению ключами.

KB2 (Класс Взаимодействия 2) – класс защиты криптографических устройств в соответствии с требованиями российских нормативных документов.

PKI (Public Key Infrastructure) – инфраструктура открытых ключей, обеспечивающая управление цифровыми сертификатами и поддерживающая работу с асимметричной криптографией.

ЭЦП (электронная цифровая подпись) – реквизит электронного документа, полученный в результате криптографического преобразования информации и предназначенный для обеспечения его подлинности и целостности.

СКЗИ (средство криптографической защиты информации) – программное или аппаратное средство, предназначенное для защиты информации с использованием криптографических методов.

HMAC (Hash-based Message Authentication Code) – код аутентификации сообщения, основанный на хэш-функции и ключе, используемый для проверки целостности данных и подлинности источника.

AES (Advanced Encryption Standard) – симметричный блочный алгоритм шифрования, принятый в качестве стандарта шифрования правительством США.

DSS (Digital Signature Standard) – стандарт цифровой подписи, определяющий алгоритмы создания и проверки ЭЦП, включая DSA.

EMV (Europay, Mastercard, and Visa) – стандарт для смарт-карт и терминалов, применяемый в банковской индустрии для безопасной обработки платежей.

SDK (Software Development Kit) – набор инструментов разработчика, включающий библиотеки, документацию и примеры кода для создания программного обеспечения.

RPC (Remote Procedure Call) – механизм удалённого вызова процедур, позволяющий программе вызывать функции в другом адресном пространстве (обычно на удалённой машине).

IDL (Interface Definition Language) – язык описания интерфейсов, применяемый для определения методов, структур и типов данных, используемых в RPC-системах.

RAII (Resource Acquisition Is Initialization) – идиома C++, при которой ресурсы (например, память, файлы) выделяются и освобождаются в пределах времени жизни объекта.

SOLID – совокупность пяти принципов объектно-ориентированного проектирования: единственная ответственность, открытость/закрытость, подстановка Барбары Лисков, разделение интерфейсов, инверсия зависимостей.

CRTP (Curiously Recurring Template Pattern) – шаблон проектирования в C++, при котором класс-потомок передаётся в шаблон в качестве параметра базовому классу.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, используемый для сериализации и десериализации структурированных данных.

Приложение Б
(рекомендуемое)

Листинг примеров работы с интерфейсами криптографических токенов

Листинг Б.1 – Использование CryptoAPI

```
#include <windows.h>
#include <wincrypt.h>
#include <iostream>
#include <vector>
#include <string_view>
#include <fmt/format.h>

int main() {
    HCRYPTPROV hProv{};
    if (!CryptAcquireContext(&hProv, nullptr, MS_ENH_RSA_AES_PROV,
        PROV_RSA_AES, CRYPT_VERIFYCONTEXT))
        return 1;

    std::string_view msg{"Hi"};
    HCRYPTHASH hHash{};
    if (CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash) &&
        CryptHashData(hHash, msg.data(), msg.size(), 0)) {
        DWORD sigLen{};
        CryptSignHash(hHash, AT_SIGNATURE, nullptr, 0, nullptr,
            &sigLen);
        std::vector<BYTE> sig(sigLen);
        if (CryptSignHash(hHash, AT_SIGNATURE, nullptr, 0,
            sig.data(), &sigLen))
            fmt::print("Signature length: {} bytes", sigLen);
        CryptDestroyHash(hHash);
    }

    CryptReleaseContext(hProv, 0);
    return 0;
}
```

Листинг Б.2 – Использование PKCS#11

```
#include <pkcs11.h>
#include <iostream>
#include <vector>
#include <string>
#include <fmt/format.h>

int main() {
    CK_RV rv{C_Initialize(nullptr)};
    if (rv != CKR_OK) {
        fmt::print(stderr, "C_Initialize failed: 0x{:X}\n", rv);
        return 1;
    }

    CK_SESSION_HANDLE session;
    rv = C_OpenSession(0, CKF_SERIAL_SESSION | CKF_RW_SESSION,
        nullptr, nullptr, &session);
    if (rv != CKR_OK) {
```

```

        fmt::print(stderr, "C_OpenSession failed: 0x{:X}\n", rv);
        return 1;
    }

    std::string_view pin{"1234"};
    rv = C_Login(session, CKU_USER, pin.data(), pin.size());
    if (rv != CKR_OK) {
        fmt::print(stderr, "C_Login failed: 0x{:X}\n", rv);
        return 1;
    }

    std::vector<CK_BYTE> data{'H','e','l','l','o',' ',
        ',','P','K','C','S','#','1','1'};
    CK_ULONG sigLen{256};
    std::vector<CK_BYTE> signature(sigLen);

    CK_MECHANISM mech{CKM_RSA_PKCS, nullptr, 0};
    CK_OBJECT_HANDLE keyHandle{1};
    rv = C_SignInit(session, &mech, keyHandle);
    if (rv != CKR_OK) {
        fmt::print(stderr, "C_SignInit failed: 0x{:X}\n", rv);
        return 1;
    }

    rv = C_Sign(session,
        data.data(), data.size(), signature.data(),
        &sigLen);
    if (rv != CKR_OK) {
        fmt::print(stderr, "C_Sign failed: 0x{:X}\n", rv);
        return 1;
    }

    signature.resize(sigLen);
    fmt::print("Signature length: {} bytes\n", sigLen);

    C_Logout(session);
    C_CloseSession(session);
    C_Finalize(nullptr);
    return 0;
}

```

Приложение В
(обязательное)
Исходный код приложения

Листинг В.1 – Файл cryptography.thrift

```
// thrift --gen cpp:no_skeleton,moveable_types -out
./src/Crypto/cryptography/api/cryptography
./src/Crypto/cryptography/idl/cryptography.thrift
// thrift --gen
py:package_prefix='crypto_dbg_util.api.cryptography.' -out
dbg_tools/crypto_dbg_util/crypto_dbg_util/api
src/Crypto/cryptography/idl/cryptography.thrift

include
"../../../../Certificates/certificate_manager/idl/certificate_manager
.thrift"

namespace cpp cryptography
namespace netstd Infotecs.VipNet.Dvcs.CryptographyApi

// Синоним типа, используемый в C++ коде
typedef binary binary_t

// Измененное и упрощенное именование включаемых структур
typedef certificate_manager.CertInfo CertificateFields
typedef certificate_manager.PubKeyInfo PublicKeyInfo

/// @brief Исключение, указывающее на ошибку в методе сервиса
exception CryptographyException {
    /// @brief Сообщение исключения
    1: required string message
}

/// @brief Представление алгоритма
struct Algorithm {
    /// @brief OID алгоритма
    1: required string oid

    /// @brief URI алгоритма
    2: required string uri

    /// @brief OID параметров
    3: optional string parameters_oid

    /// @brief Описание
    /// @note Хранится в пригодном для отображения виде
    4: optional string description
}

/// @brief Статус профиля
struct ProfileStatus {
    /// @brief Доступность данного профиля
    1: required bool available

    /// @brief Аутентифицированность пользователя
```

```

    /// @details true, если пользователь аутентифицирован, иначе
false
    2: required bool logged_in

    /// @brief Доступность функции подписи
    3: required bool sign_available

    /// @brief Доступность функции проверки подписи
    4: required bool verify_available

    /// @brief Доступность функции хеширования
    5: required bool digest_available
}

/// @brief Статус службы
struct Status {
    /// @brief Доступность поставщика криптографических операций
    1: required bool available

    /// @brief Соответствие профиля и его статуса
    /// @note Ключом является название профиля
    2: optional map<string, ProfileStatus> profile_statuses
}

/// Далее определены структуры для входных и выходных параметров
методов сервиса

/// @brief Входные параметры функции `Sign`
struct SignRequest {
    /// @brief Профиль режима работы
    1: required string profile

    /// @brief Хеш сообщения
    2: required binary_t data
}

/// @brief Выходные параметры функции `Sign`
struct SignResponse {
    /// @brief Подпись для сообщения
    1: required binary_t signature
}

/// @brief Входные параметры функции `Verify`
struct VerifyRequest {
    /// @brief Профиль режима работы
    1: required string profile

    /// @brief Хеш сообщения
    2: required binary_t data

    /// @brief Подпись
    3: required binary_t signature
}

```

```

    /// @brief Описание открытого ключа
    4: required PublicKeyInfo pki
}

/// @brief Выходные параметры функции `Verify`
struct VerifyResponse {
    /// @brief Результат проверки подписи
    /// @details true, если подпись верна, иначе false
    1: required bool result
}

/// @brief Входные параметры функции `Digest`
struct DigestRequest {
    /// @brief Профиль режима работы
    1: required string profile

    /// @brief Сообщение
    2: required binary_t data

    /// @brief OID алгоритма хеширования
    3: required string oid
}

/// @brief Выходные параметры функции `Digest`
struct DigestResponse {
    /// @brief Хеш-значение
    1: required binary_t digest
}

/// @brief Входные параметры функции `GetSignAlgorithm`
struct GetSignAlgorithmRequest {
    /// @brief Профиль режима работы
    1: required string profile
}

/// @brief Выходные параметры функции `GetSignAlgorithm`
struct GetSignAlgorithmResponse {
    /// @brief Представление алгоритма подписи
    1: required Algorithm algorithm
}

/// @brief Входные параметры функции `GetDigestAlgorithm`
struct GetDigestAlgorithmRequest {
    /// @brief Профиль режима работы
    1: required string profile
}

/// @brief Выходные параметры функции `GetDigestAlgorithm`
struct GetDigestAlgorithmResponse {
    /// @brief Представление алгоритма хеширования
    1: required Algorithm algorithm
}

```

```

}

/// @brief Входные параметры функции `GetCertificate`
struct GetCertificateRequest {
    /// @brief Профиль режима работы
    1: required string profile
}

/// @brief Выходные параметры функции `GetCertificate`
struct GetCertificateResponse {
    /// @brief Бинарные данные используемого сертификата
    1: required binary_t data

    /// @brief Бинарные данные других сертификатов в цепочке
    2: optional list<binary_t> chain

    /// @brief Поля используемого сертификата
    3: required CertificateFields fields
}

/// @brief Выходные параметры функции `GetStatus`
struct GetCryptographyStatusResponse {
    /// @brief Статус службы
    1: required Status status
}

/// @brief Целевые методы службы криптографии
service Cryptography {
    /// @brief Создание подписи
    /// @param request Структура с входными параметрами
    /// @returns SignResponse Структура с выходными параметрами
    /// @throws CryptographyException
    SignResponse Sign (
        1: required SignRequest request
    ) throws (
        1: CryptographyException exception
    )

    /// @brief Проверка подписи
    /// @param request Структура с входными параметрами
    /// @returns VerifyResponse Структура с выходными параметрами
    /// @throws CryptographyException
    VerifyResponse Verify (
        1: required VerifyRequest request
    ) throws (
        1: CryptographyException exception
    )

    /// @brief Вычисление хеш-значения
    /// @param request Структура с входными параметрами
    /// @returns DigestResponse Структура с выходными параметрами
    /// @throws CryptographyException

```

```

DigestResponse Digest (
    1: required DigestRequest request
) throws (
    1: CryptographyException exception
)

/// @brief Получение используемого алгоритма подписи
/// @param request Структура с входными параметрами
/// @returns GetSignAlgorithmResponse Структура с выходными
параметрами
/// @throws CryptographyException
GetSignAlgorithmResponse GetSignAlgorithm (
    1: required GetSignAlgorithmRequest request
) throws (
    1: CryptographyException exception
)

/// @brief Получение используемого алгоритма хеширования
/// @param request Структура с входными параметрами
/// @returns GetDigestAlgorithmResponse Структура с выходными
параметрами
/// @throws CryptographyException
GetDigestAlgorithmResponse GetDigestAlgorithm (
    1: required GetDigestAlgorithmRequest request
) throws (
    1: CryptographyException exception
)

/// @brief Получение используемого сертификата
/// @param request Структура с входными параметрами
/// @returns GetCertificateResponse Структура с выходными
параметрами
/// @throws CryptographyException
GetCertificateResponse GetCertificate (
    1: required GetCertificateRequest request
) throws (
    1: CryptographyException exception
)

/// @brief Получение статуса службы
/// @returns GetCryptographyStatusResponse Структура с
выходными параметрами
/// @throws CryptographyException
GetCryptographyStatusResponse GetStatus() throws (
    1: CryptographyException exception
)
}

```

Листинг В.2 – Файл control.thrift

```

// thrift --gen cpp:no_skeleton,moveable_types -out
./src/Crypto/cryptography/api/control
./src/Crypto/cryptography/idl/control.thrift

```

```
// thrift --gen py:package_prefix='crypto_dbg_util.api.control.' -
out dbg_tools/crypto_dbg_util/crypto_dbg_util/api
src/Crypto/cryptography/idl/control.thrift
// На выходе получится сломанный трифт-клиент, в котором надо
починить руками десятки импортов!

include
"../../../../Certificates/certificate_manager/idl/certificate_manager
.thrift" # Вот так делать НЕЛЬЗЯ! Это приводит к проблемам при
генерации клиентов!
include "cryptography.thrift" # И вот так - тоже нельзя

namespace cpp cryptography.control
namespace netstd Infotecs.VipNet.Dvcs.CryptographyControlApi

// Синоним типа, используемый в C++ коде
typedef binary binary_t

// Упрощенное именование включаемых структур
typedef certificate_manager.CertInfo CertificateFields
typedef cryptography.Algorithm Algorithm
typedef cryptography.Status Status

/// @brief Тип хранилища сертификатов CryptoApi.
enum CertStore {
    CERT_STORE_PROV_SYSTEM_CURRENT_USER = 0,
    CERT_STORE_PROV_SYSTEM_LOCAL_MACHINE = 1,
}

/// @brief Набор параметров для режима работы через OpenSSL
struct OpenSslProfile {
    /// @brief Путь к файлу сертификата
    /// @note При отсутствии используется по умолчанию
    1: optional string certificate

    /// @brief Путь к файлу закрытого ключа
    /// @note При отсутствии используется по умолчанию
    2: optional string key
}

/// @brief Набор параметров для режима работы через CryptoApi
struct CryptoApiProfile {
    /// @brief Тип хранилища сертификатов CryptoApi.
    1: optional CertStore cert_store
    /// @brief Пароль для доступа к ключу
    /// @warning Не возвращается
    2: optional string password
}

/// @brief Набор параметров для режима работы через PKCS#11
struct PkcsProfile {
    /// @brief Идентификатор токена
```

```

    /// @note Используется значение, определенное в структуре
    `Device`
    1: required binary_t device_identity

    /// @brief Пароль, аутентифицирующий на токене
    /// @warning Не сохраняется и не возвращается
    2: optional string token_password
}

/// @brief Набор параметров для режима работы через ЭЦП ДТС ЕАЭС
/// @note Набор настроек совпадает с pkcs
typedef PkcsProfile TtpHsmProfile

/// @brief Объединение наборов параметров для режимов работы
union RegimeProfile {
    /// @brief Набор параметров для режима работы через PKCS#11
    1: optional PkcsProfile pkcs

    /// @brief Набор параметров для режима работы через ЭЦП ДТС
    ЕАЭС
    2: optional TtpHsmProfile ttp_hsm

    /// @brief Набор параметров для режима работы через CryptoApi
    3: optional CryptoApiProfile crypto_api

    /// @brief Набор параметров для режима работы через OpenSSL
    4: optional OpenSslProfile open_ssl
}

/// @brief Профиль использования поставщика
struct Profile {
    /// @brief Алгоритм хеширования
    /// @note Если не указать, используется ГОСТ Р 34.11-2012 512
    бит
    /// @note Заданный алгоритм не используется при вызове
    операции хеширования
    1: optional Algorithm digest

    /// @brief Бинарные данные используемого сертификата
    2: optional binary_t certificate_data

    /// @brief Бинарные данные других сертификатов в цепочке
    3: optional list<binary_t> certificate_chain

    /// @brief Поля используемого сертификата
    4: optional CertificateFields certificate_fields

    /// @brief Набор параметров для режима работы
    5: required RegimeProfile regime_profile
}

/// @brief Режим работы через OpenSsl

```

```

struct OpenSsl { }

/// @brief Режим работы через CryptoApi
struct CryptoApi { }

/// @brief Режим работы через ЭЦП ДТС ЕАЭС
struct TtpHsm {
    /// @brief IP-адрес ЭЦП ДТС ЕАЭС
    1: required string host
}

/// @brief Режим работы через PKCS#11
struct Pkcs { }

/// @brief Объединение режимов работы
union Regime {
    /// @brief Режим работы через PKCS#11
    1: optional Pkcs pkcs

    /// @brief Режим работы через ЭЦП ДТС ЕАЭС
    2: optional TtpHsm ttp_hsm

    /// @brief Режим работы через CryptoApi
    3: optional CryptoApi crypto_api

    /// @brief Режим работы через OpenSsl
    4: optional OpenSsl open_ssl
}

/// @brief Поставщик криптографических операций
struct Provider {
    /// @brief Наименование поставщика
    /// @note Хранится в пригодном для отображения виде
    1: required string name

    /// @brief Описание поставщика
    /// @note Хранится в пригодном для отображения виде
    2: optional string description

    /// @brief Режим работы
    3: required Regime regime
}

/// @brief Настройки службы криптографии
struct Settings {
    /// @brief Поставщик криптографических операций
    /// @note Отсутствие поля указывает на его неизменность при
    выполнении метода `SetSettings`
    1: optional Provider provider

    /// @brief Доступные профили
    /// @note Ключом является название профиля

```

```

    2: optional map<string, Profile> profiles

    /// @brief Метка времени активной конфигурации
    /// @note Формат: Unix time в миллисекундах
    3: required i64 timestamp
}

/// @brief Настройки подключения к удаленной службе
struct RemoteSettings {
    /// @brief IP-адрес удалённого сервиса
    1: required string host

    /// @brief Порт функций криптографии
    2: required i16 cryptography_port

    /// @brief Порт функций управления
    3: required i16 control_port

    /// @brief Метка времени настроек подключения к удаленной
    службе
    /// @note Формат: Unix time в миллисекундах
    4: required i64 timestamp
}

/// @brief Устройство подписи
/// @note В терминах PKCS#11 - токен
struct Device {
    /// @brief Идентификатор устройства
    1: required binary_t identity

    /// @brief Описание устройства
    /// @note Хранится в пригодном для отображения виде
    2: optional string description
}

/// Далее определены структуры для входных и выходных параметров
методов сервиса

/// @brief Входные параметры функции `SetSettings`
struct SetSettingsRequest {
    /// @brief Настройки службы криптографии
    1: required Settings settings
}

/// @brief Входные параметры функции `SetRemoteSettings`
struct SetRemoteSettingsRequest {
    /// @brief Настройки подключения к удаленной службе
    1: required RemoteSettings remote_settings
}

/// @brief Входные параметры функции `CheckSettings`
struct CheckSettingsRequest {

```

```

    /// @brief Настройки службы криптографии
    1: required Settings settings

    /// @brief Направлен ли данный вызов к удаленной службе
    2: optional bool remote
}

/// @brief Выходные параметры функции `CheckSettings`
struct CheckSettingsResponse {
    /// @brief Статус, который возможен, если настройки применены
    1: required Status status
}

/// @brief Входные параметры функции `CheckRemoteSettings`
struct CheckRemoteSettingsRequest {
    /// @brief Настройки подключения к удаленной службе
    /// @note В случае отсутствия проверяются текущие настройки
    подключения к удаленной службе
    1: optional RemoteSettings remote_settings
}

/// @brief Выходные параметры функции `GetSettings`
struct GetSettingsResponse {
    /// @brief Активные настройки
    1: required Settings settings
}

/// @brief Выходные параметры функции `GetRemoteSettingsStatus`
struct GetRemoteSettingsResponse {
    /// @brief Настройки подключения к удаленной службе
    /// @note При отсутствии подключения к удаленной службе
    настройки не возвращаются
    1: optional RemoteSettings remote_settings
}

/// @brief Выходные параметры функции `GetProviders`
struct GetProvidersResponse {
    /// @brief Поставщики криптографических операций
    1: required list<Provider> providers
}

/// @brief Выходные параметры функции `GetDevices`
struct GetDevicesResponse {
    /// @brief Устройства подписи
    1: required list<Device> devices
}

/// @brief Выходные параметры функции `GetAlgorithms`
struct GetAlgorithmsResponse {
    /// @brief Алгоритмы хеширования
    1: optional list<Algorithm> digest
}

```

```

    /// @brief Алгоритмы подписи
    2: optional list<Algorithm> sign
}

/// @brief Выходные параметры функции `GetStatus`
struct GetControlStatusResponse {
    /// @brief Состояние службы
    1: required Status status

    /// @brief Метка времени данной конфигурации
    /// @note Формат: Unix time в миллисекундах
    2: required i64 timestamp
}

/// @brief Выходные параметры функции `GetRemoteStatus`
struct GetRemoteStatusResponse {
    /// @brief Метка времени настроек подключения к удаленной
    службе
    /// @note Формат: Unix time в миллисекундах
    1: required i64 timestamp

    /// @brief Включен ли режим удаленного подключения
    2: required bool enabled

    /// @brief Доступна ли удаленная служба
    3: required bool available
}

/// Алгоритм инициализации

/// Инициализация локального режима
/// 1. Разъединение удаленного подключения средствами
CloseRemoteConnection
/// 2. Запрос списка поддерживаемых поставщиков средствами
GetProviders
/// 3. Установка поставщика (только поставщика через SetSettings)
/// 4. Настройка поставщика:
///   Для режима Pkcs
///   4.1. Поле `available` структуры `GetStatusResponse` теперь
`true`, следовательно, доступны функции `GetDevices` и
`GetAlgorithms`
///   4.2. Используя данные функции, можно заполнить поле `Pkcs`
через `SetSettings`
///   4.3. Через GetStatus можно узнавать, что получилось

/// Инициализация удаленного режима
/// 1. Установка настроек через SetRemoteSettings
/// 2. Поле `available` структуры `GetRemoteStatusResponse` теперь
`true`, следовательно, все функции обращаются к удаленной службе
/// 3. Оставшиеся действия выполняются так же, как в локальном
режиме, начиная со второго шага

```

```

/// @brief Интерфейс локальных операций управления сервисом
криптографии
service ControlLocal {
    /// @brief Применение настроек для службы криптографии
    /// @param request Структура с входными параметрами
    /// @throws cryptography.CryptographyException
    void SetSettings(
        1: required SetSettingsRequest request
    ) throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Выполнение проверки настроек без применения
    /// @note Возможности проверки ограничены
    /// @param request Структура с входными параметрами
    /// @returns CheckSettingsResponse Структура с выходными
параметрами
    /// @throws cryptography.CryptographyException
    CheckSettingsResponse CheckSettings(
        1: required CheckSettingsRequest request
    ) throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение текущих активных настроек
    /// @returns GetSettingsResponse Структура с выходными
параметрами
    /// @throws cryptography.CryptographyException
    GetSettingsResponse GetSettings() throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение списка поставщиков криптографических
операций
    /// @returns GetProvidersResponse Структура с выходными
параметрами
    /// @throws cryptography.CryptographyException
    GetProvidersResponse GetProviders() throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение списка доступных устройств
    /// @note Вызов доступен не во всех режимах и может быть
произведен только при Status::available == true
    /// @returns GetDevicesResponse Структура с выходными
параметрами
    /// @throws cryptography.CryptographyException
    GetDevicesResponse GetDevices() throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение списка поддерживаемых алгоритмов

```

```

    /// @note Вызов доступен не во всех режимах и может быть
    произведен только при Status::available == true
    /// @returns GetAlgorithmsResponse Структура с выходными
    параметрами
    /// @throws cryptography.CryptographyException
    GetAlgorithmsResponse GetAlgorithms() throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение статуса службы
    /// @returns GetControlStatusResponse Структура с выходными
    параметрами
    /// @throws cryptography.CryptographyException
    GetControlStatusResponse GetStatus() throws (
        1: cryptography.CryptographyException exception
    )
}

/// @brief Интерфейс локальных и удаленных операций управления
сервисом криптографии
service Control extends ControlLocal {
    /// @brief Установка настроек подключения к удаленной службе
    /// @param request Структура с входными параметрами
    /// @throws cryptography.CryptographyException
    void SetRemoteSettings(
        1: required SetRemoteSettingsRequest request
    ) throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Проверка настроек подключения к удаленной службе
    /// @param request Структура с входными параметрами
    /// @throws cryptography.CryptographyException
    void CheckRemoteSettings(
        1: required CheckRemoteSettingsRequest request
    ) throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Разъединение подключения к удаленной службе
    /// @throws cryptography.CryptographyException
    void CloseRemoteConnection() throws (
        1: cryptography.CryptographyException exception
    )

    /// @brief Получение настроек подключения к удаленной службе
    /// @returns GetRemoteSettingsResponse Структура с выходными
    параметрами
    /// @throws cryptography.CryptographyException
    GetRemoteSettingsResponse GetRemoteSettings() throws (
        1: cryptography.CryptographyException exception
    )
}

```

```

    /// @brief Получение статуса удаленного подключения
    /// @returns GetRemoteStatusResponse Структура с выходными
    параметрами
    /// @throws cryptography.CryptographyException
    GetRemoteStatusResponse GetRemoteStatus() throws (
        1: cryptography.CryptographyException exception
    )
}

```

Листинг В.3 – Файл settings_tests.cpp

```

#include <gtest/gtest.h>

#include <nlohmann/json.hpp>

#include "settings.hpp"

using namespace cryptography;
using json = nlohmann::json;

class EndpointTest : public testing::Test
{
protected:
    void SetUp() final {};

    void TearDown() final {};
};

TEST_F(EndpointTest, Endpoint_Normalize_EmptyPath)
{
    Endpoint endpoint{};
    endpoint.Normalize({.path = "/tmp/cryptography.sock"});
    EXPECT_EQ(endpoint, (Endpoint{.path =
"/tmp/cryptography.sock"}));
}

TEST_F(EndpointTest, Endpoint_Normalize_InvalidHighPort)
{
    Endpoint endpoint{.port = 80000};
    endpoint.Normalize({.port = 10000});
    EXPECT_EQ(endpoint, (Endpoint{.port = 10000}));
}

TEST_F(EndpointTest, Endpoint_Normalize_NegativePort)
{
    Endpoint endpoint{.port = -80000};
    endpoint.Normalize({.port = 10000});
    EXPECT_EQ(endpoint, (Endpoint{.port = 10000}));
}

TEST_F(EndpointTest, Endpoint_Normalize_EmptyIp)
{

```

```

        Endpoint endpoint{.ip = "", .port = 10000};
        endpoint.Normalize({});
        EXPECT_EQ(endpoint, (Endpoint{.ip = "127.0.0.1", .port =
10000}));
    }

TEST_F(EndpointTest, Endpoint_Normalize_DoesNotOverwriteValidPort)
{
    Endpoint endpoint{.port = 8080};
    endpoint.Normalize({.port = 9090});
    EXPECT_EQ(endpoint, (Endpoint{.port = 8080}));
}

TEST_F(EndpointTest, Endpoint_ConnectionString_PathPreferred)
{
    const Endpoint endpoint{.path = "/run/cryptography.sock", .ip
= "10.0.0.0", .port = 10000};
    EXPECT_EQ(endpoint.ConnectionString(),
"/run/cryptography.sock");
}

TEST_F(EndpointTest, Endpoint_ConnectionString_IpPort)
{
    const Endpoint endpoint{.ip = "10.0.0.0", .port = 10000};
    EXPECT_EQ(endpoint.ConnectionString(), "10.0.0.0:10000");
}

TEST_F(EndpointTest, Endpoint_ConnectionString_DefaultValues)
{
    const Endpoint endpoint{};
    EXPECT_EQ(endpoint.ConnectionString(), "127.0.0.1:0");
}

TEST_F(EndpointTest, Endpoint_ConnectionString_Ipv6)
{
    const Endpoint endpoint{.ip = "::1", .port = 10000};
    EXPECT_EQ(endpoint.ConnectionString(), "::1:10000");
}

class CryptographySettingsTest : public testing::Test
{
protected:
    void SetUp() final {}

    void TearDown() final {}
};

TEST_F(CryptographySettingsTest,
CryptographySettings_Normalize_RewritesOutOfRangePorts)
{
    CryptographySettings settings{.control{.port = -10000},
.cryptography{.port = -10000}, .diagnostic{.port = 80000}};

```

```

        settings.Normalize();
        EXPECT_EQ(settings,
                    (CryptographySettings{.control{.port = 3334},
                    .cryptography{.port = 3335}, .diagnostic{.port = 3336}}));
    }

    TEST_F(CryptographySettingsTest,
    CryptographySettings_Normalize_DoesNotTouchValidPorts)
    {
        CryptographySettings settings{.control{.port = 10000},
        .cryptography{.port = 10000}, .diagnostic{.port = 10000}};
        settings.Normalize();
        EXPECT_EQ(settings, (CryptographySettings{
                            .control{.port = 10000},
        .cryptography{.port = 10000}, .diagnostic{.port = 10000}}));
    }

    TEST_F(CryptographySettingsTest,
    CryptographySettings_Instance_IsSingleton)
    {
        const auto& first_instance{CryptographySettings::Instance()};
        const auto& second_instance{CryptographySettings::Instance()};
        EXPECT_EQ(&first_instance, &second_instance);
    }

    TEST_F(CryptographySettingsTest,
    CryptographySettings_Instance_ReinitializeRestoresDefaults)
    {
        auto& first_instance{CryptographySettings::Instance()};
        first_instance.control = Endpoint{.port = 10000};
        const auto&
        second_instance{CryptographySettings::Instance(true)};
        EXPECT_EQ(&first_instance, &second_instance);
        EXPECT_EQ(second_instance.control.port, 3334);
    }

    TEST_F(CryptographySettingsTest,
    CryptographySettings_Read_FromStream)
    {
        const auto
        content{R"({"control":{"port":10000},"cryptography":{"port":10000}
        ,"diagnostic":{"port":10000}})"};

        CryptographySettings settings{};
        ASSERT_TRUE(settings.Read(std::istringstream{content}));
        EXPECT_EQ(settings, (CryptographySettings{
                            .control{.port = 10000},
        .cryptography{.port = 10000}, .diagnostic{.port = 10000}}));
    }

```

Листинг В.4 – Файл control_handler_tests.cpp

```
#include <gmock/gmock.h>
```

```

#include <gtest/gtest.h>

#include "control_handler.hpp"
#include "handler_pool.hpp"

using namespace cryptography;
using namespace testing;

class MockHandlerPool final : public HandlerPool
{
public:
    MOCK_METHOD(void, Reset, (provider_t, const profiles_t&));
    MOCK_METHOD(Status, GetStatus, (), (const));
    MOCK_METHOD(std::vector<Device>, GetDevices, (), (const));
};

class ControlHandlerTest : public Test
{
protected:
    void SetUp() final
    {
        pool_ = std::make_shared<MockHandlerPool>();
        handler_ = std::make_unique<ControlHandler>(pool_);
    }

    std::shared_ptr<MockHandlerPool> pool_{};

    std::unique_ptr<ControlHandler> handler_{};
};

TEST_F(ControlHandlerTest,
ControlHandler_SetSettings_SavesSettingsAndResetsPool)
{
    Regime regime{};
    regime.__set_pkcs({});

    Provider provider{};
    provider.__set_name("pkcs11");
    provider.__set_regime(regime);

    Settings settings{};
    settings.__set_provider(provider);

    SetSettingsRequest request{};
    request.__set_settings(settings);

    EXPECT_CALL(*pool_, Reset(_, _)).Times(1);

    handler_->SetSettings(request);

    GetSettingsResponse response{};
    handler_->GetSettings(response);
}

```

```

        EXPECT_EQ(response.settings.provider.name, "pkcs11");
    }

    TEST_F(ControlHandlerTest,
ControlHandler_GetDevices_DelegatesToPool)
    {
        Device first_device{};
        first_device.__set_identity("token");
        Device second_device{};
        second_device.__set_identity("token");
        const std::vector<Device> devices{first_device,
second_device};
        EXPECT_CALL(*pool_, GetDevices()).WillOnce(Return(devices));

        GetDevicesResponse response{};
        handler_>GetDevices(response);

        EXPECT_EQ(response.devices, devices);
    }

    TEST_F(ControlHandlerTest,
ControlHandler_GetStatus_DelegatesAndReturnsTimestamp)
    {
        Status status{};
        EXPECT_CALL(*pool_, GetStatus()).WillOnce(Return(status));

        GetControlStatusResponse response{};
        handler_>GetStatus(response);

        EXPECT_EQ(response.status, status);
        EXPECT_EQ(response.timestamp, 0);
    }

    TEST_F(ControlHandlerTest,
ControlHandler_SetSettings_InfocomSuccess)
    {
        const json request =

R"({"settings":{"profiles":{},"provider":{"description":"","name":
"", "regime":{"infocom":null}}, "timestamp":0}})";

        handler_>SetSettings(request);
    }

```

Приложение Г
(рекомендуемое)

Диаграммы классов сгенерированного кода интерфейсов

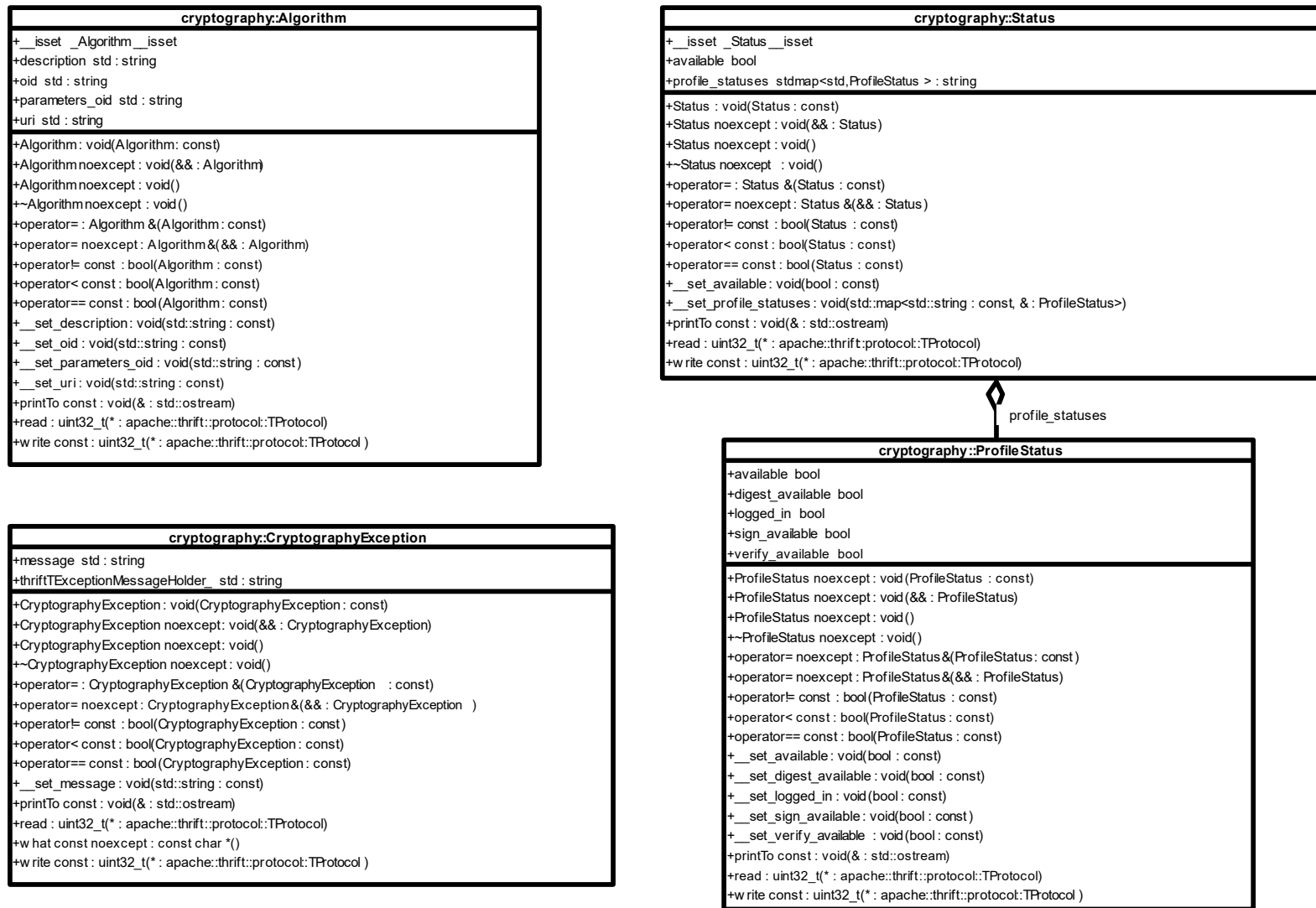


Рисунок Г.1 – Диаграмма классов моделей данных сервиса криптографических операций

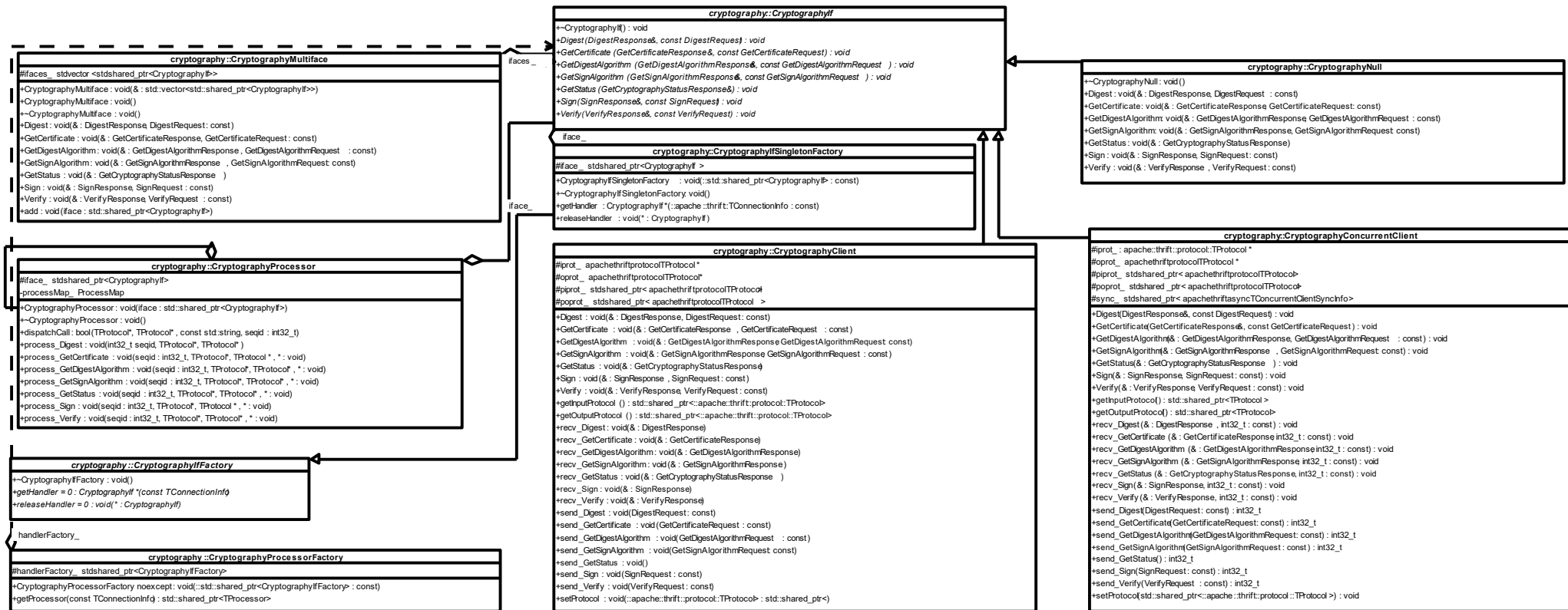
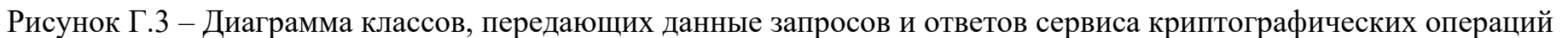


Рисунок Г.2 – Диаграмма классов инфраструктуры Apache Thrift сервиса криптографических операций



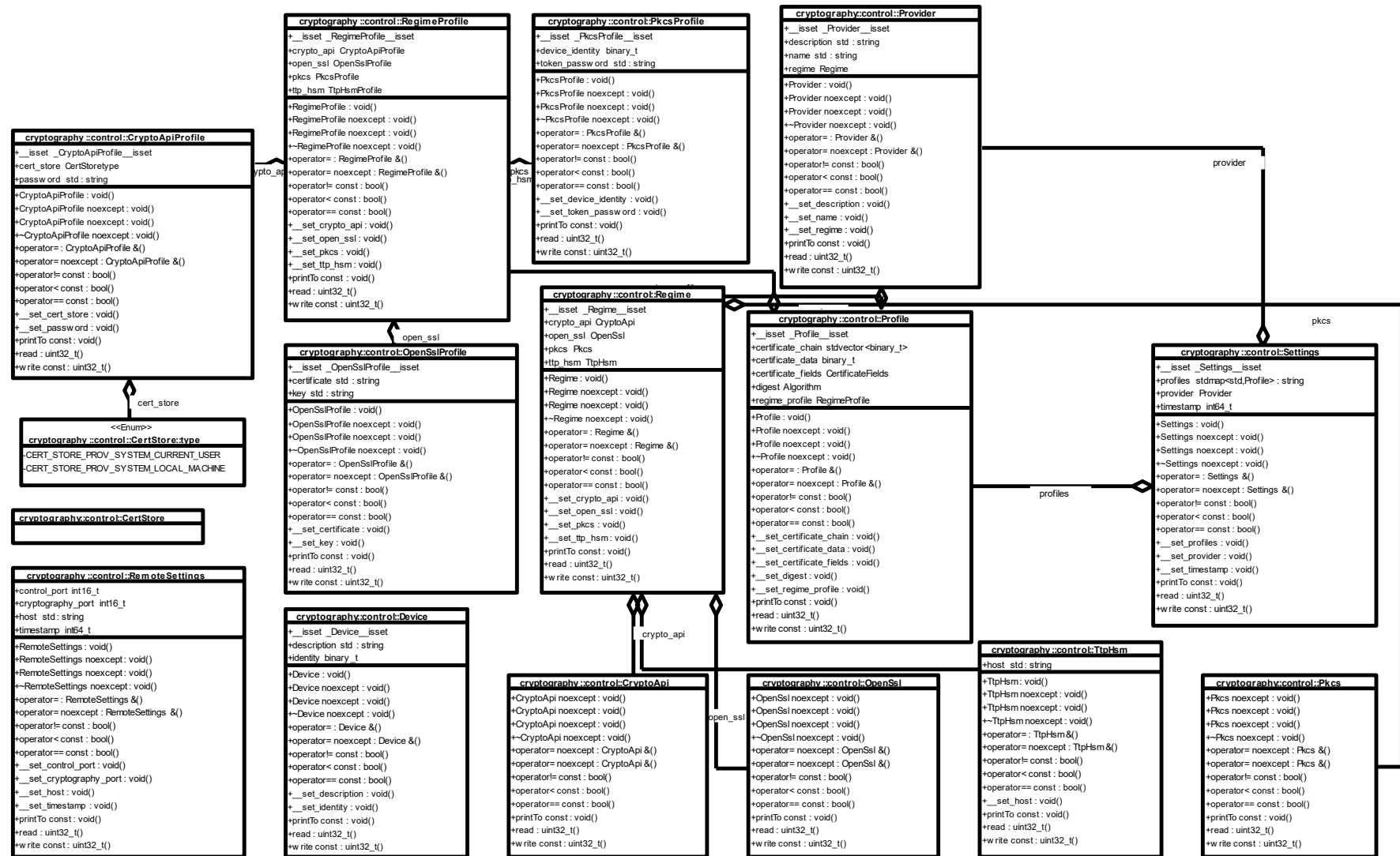


Рисунок Г.4 – Диаграмма классов моделей данных сервиса служебных операций

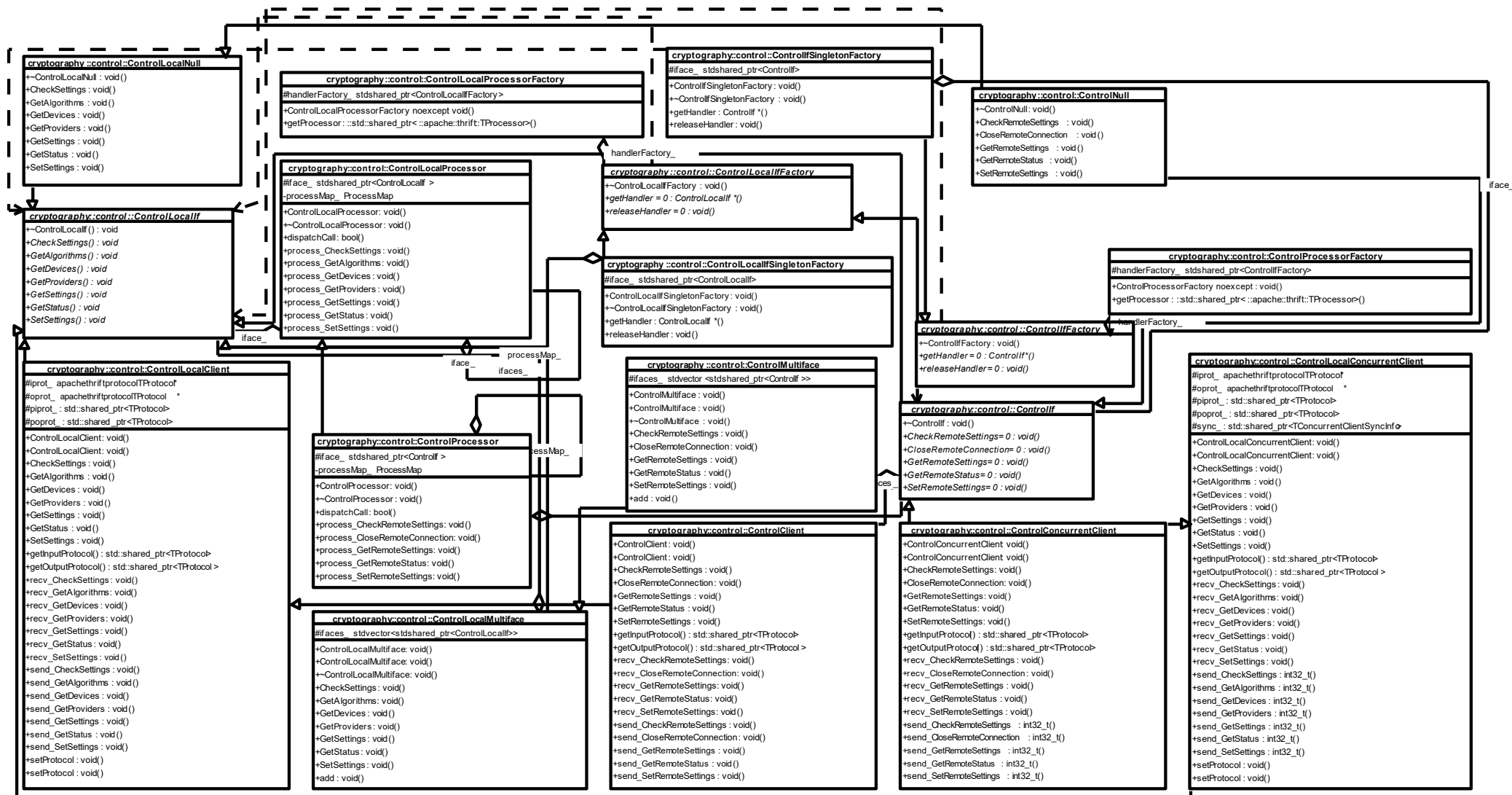


Рисунок Г.5 – Диаграмма классов инфраструктуры Apache Thrift сервиса служебных операций

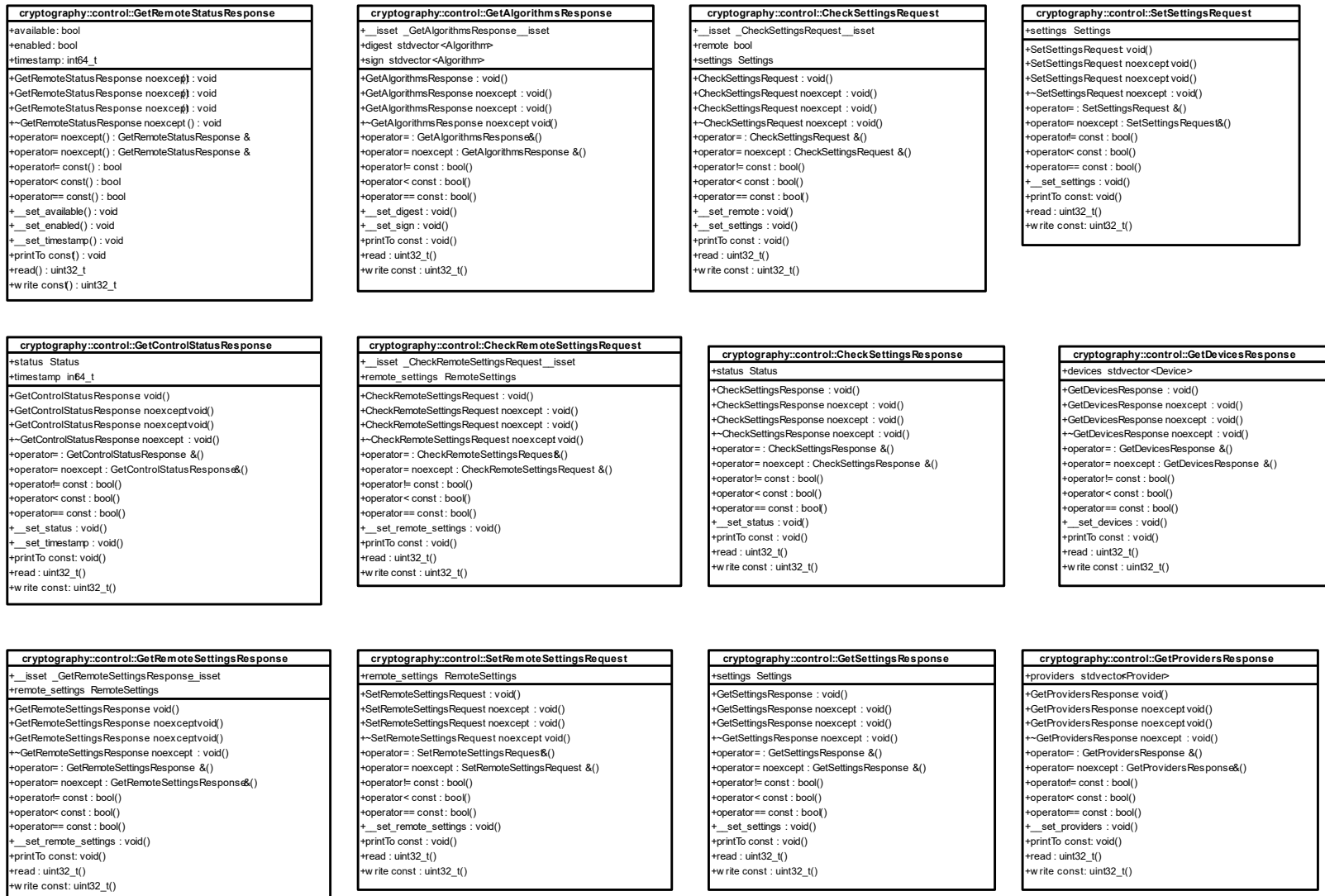


Рисунок Г.6 – Диаграмма классов, передающих данные запросов и ответов сервиса служебных операций

Приложение Д
(рекомендуемое)

Диаграммы классов разработанного приложения

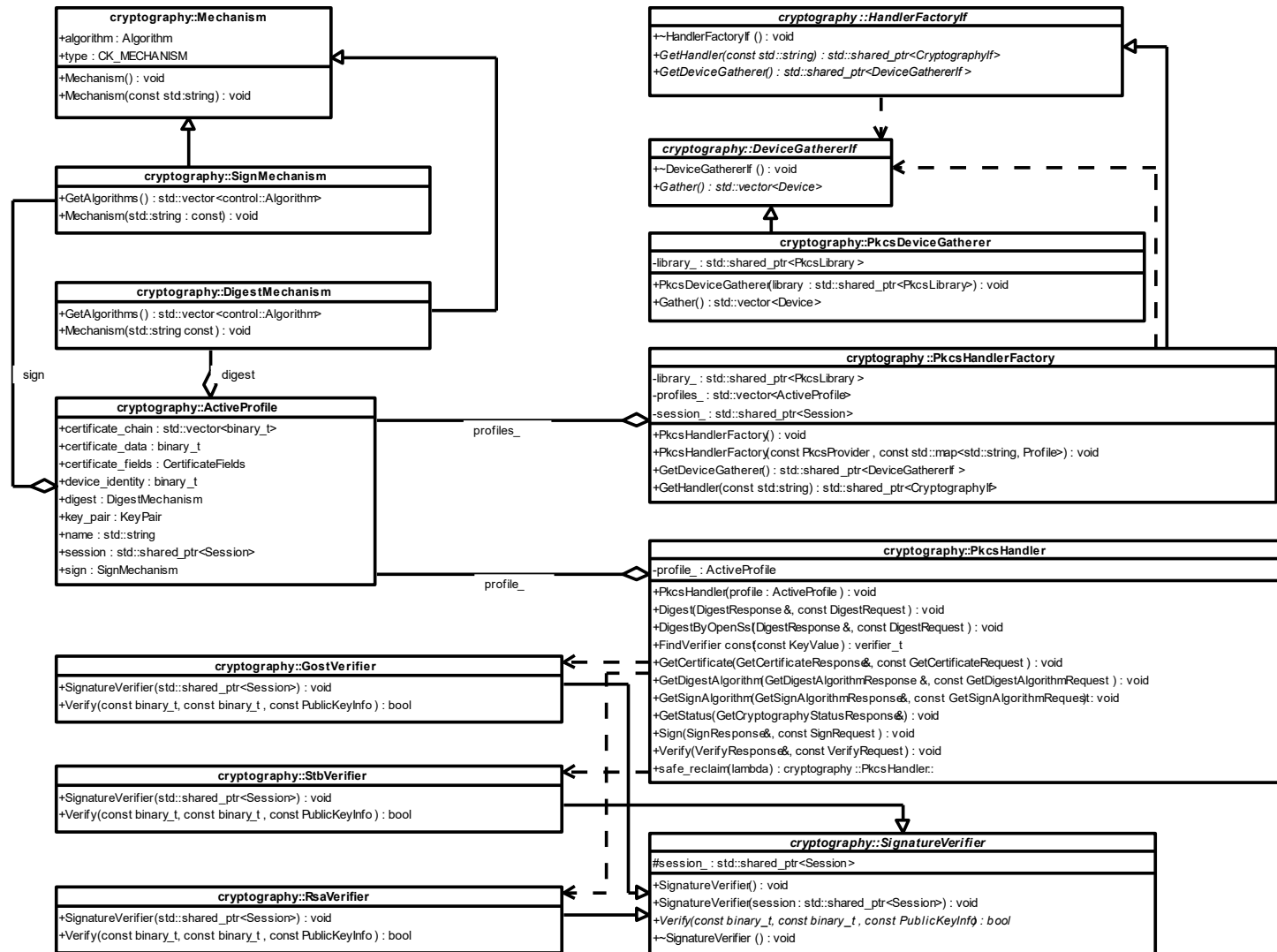


Рисунок Д.1 – Диаграмма классов основной логики выполнения криптографических операций

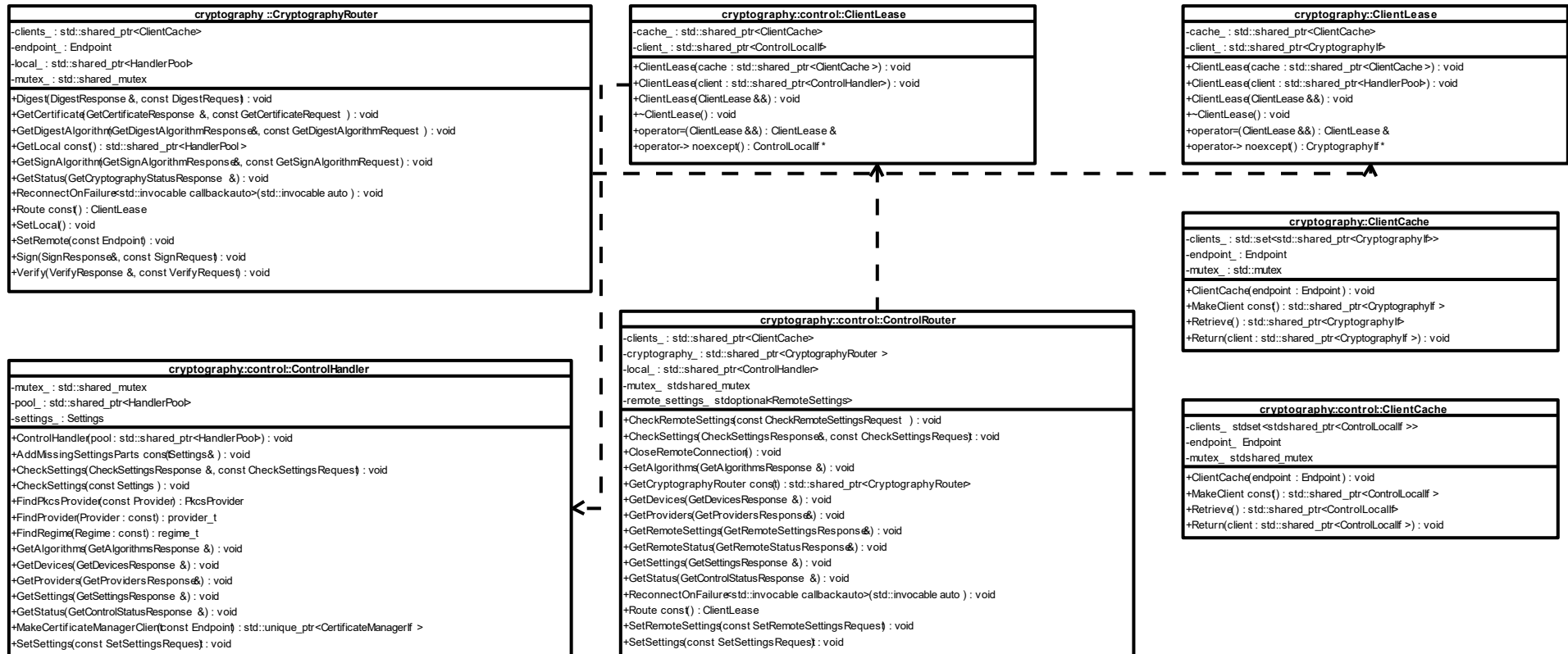


Рисунок Д.2 – Диаграмма классов управления и маршрутизации вызовов

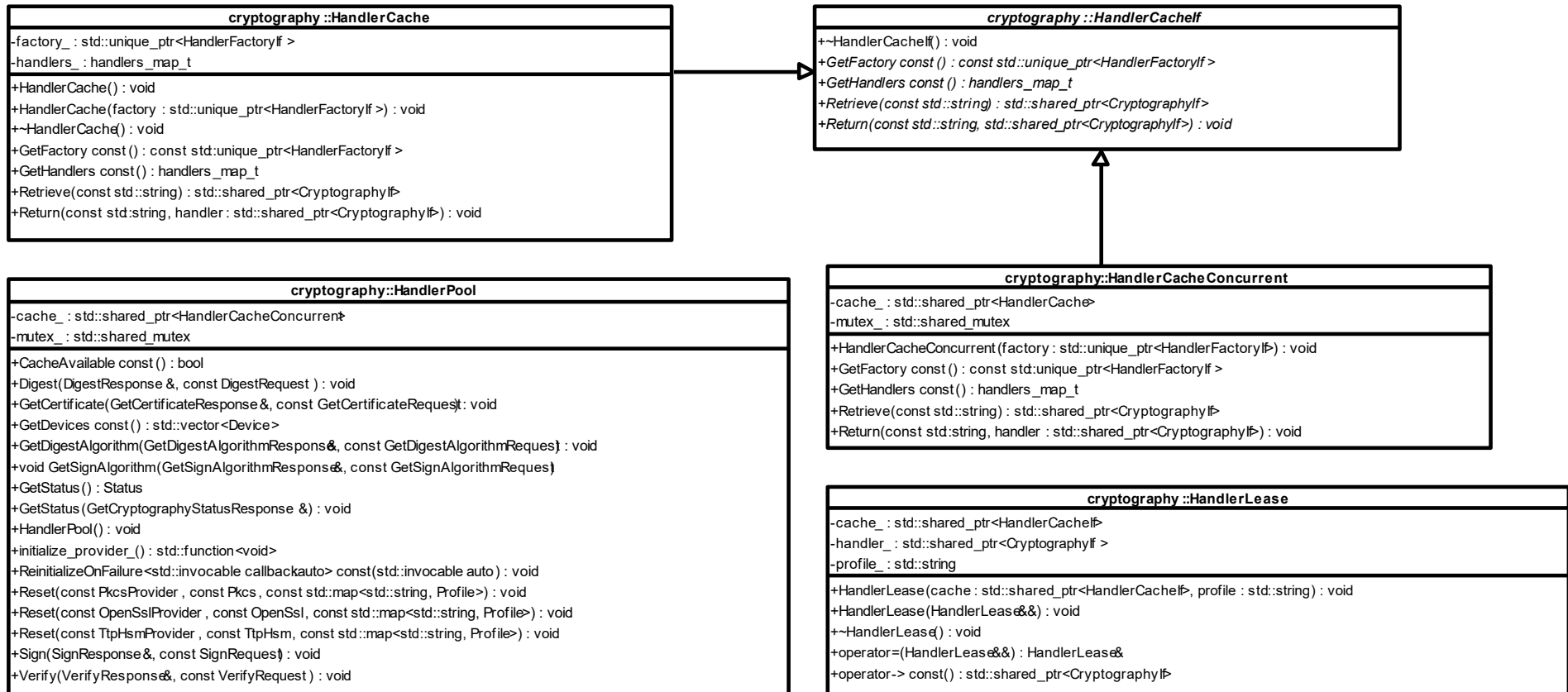


Рисунок Д.3 – Диаграмма классов кеширования обработчиков вызовов

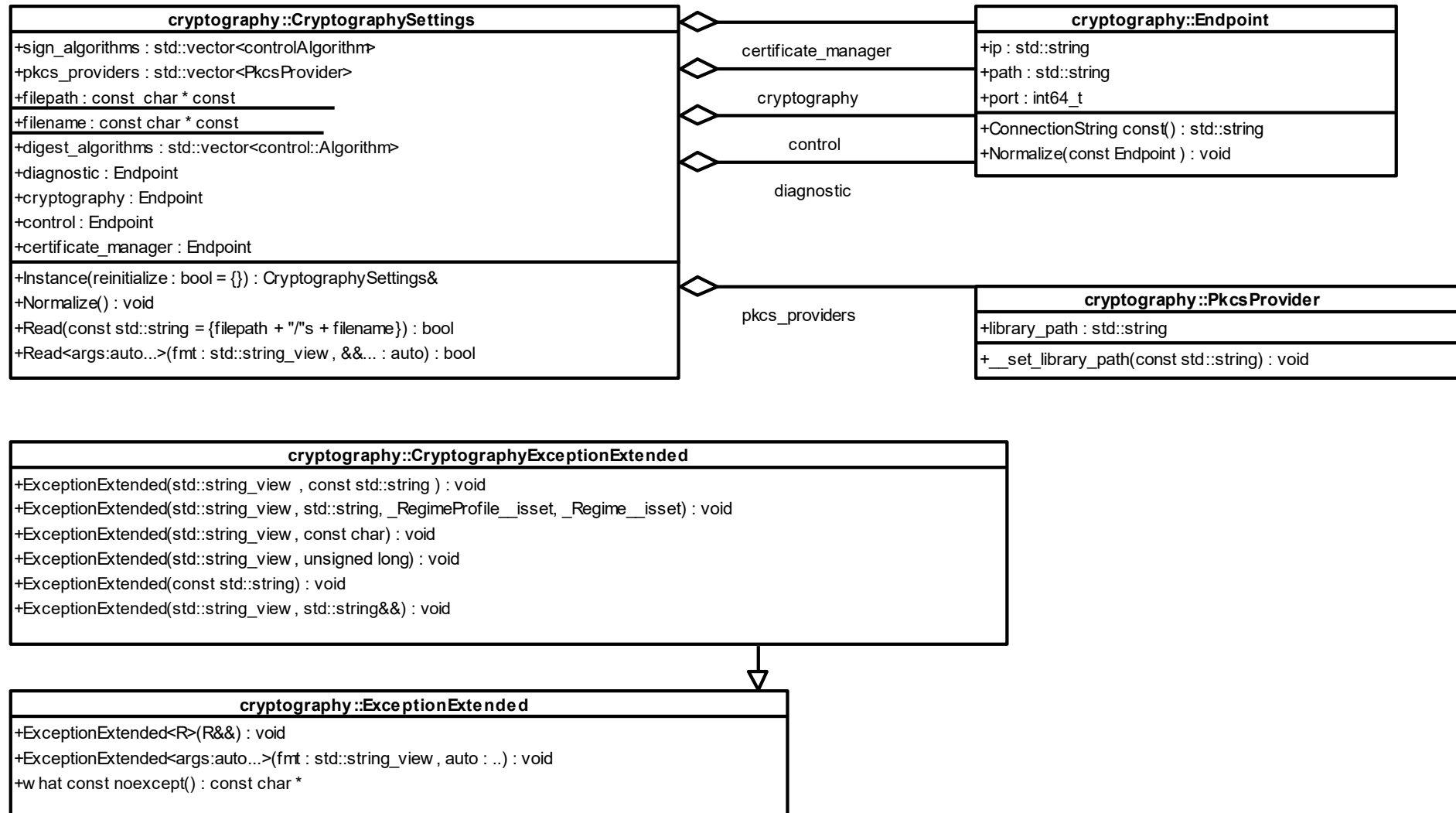


Рисунок Д.4 – Диаграмма классов конфигурации и обработки исключительных ситуаций

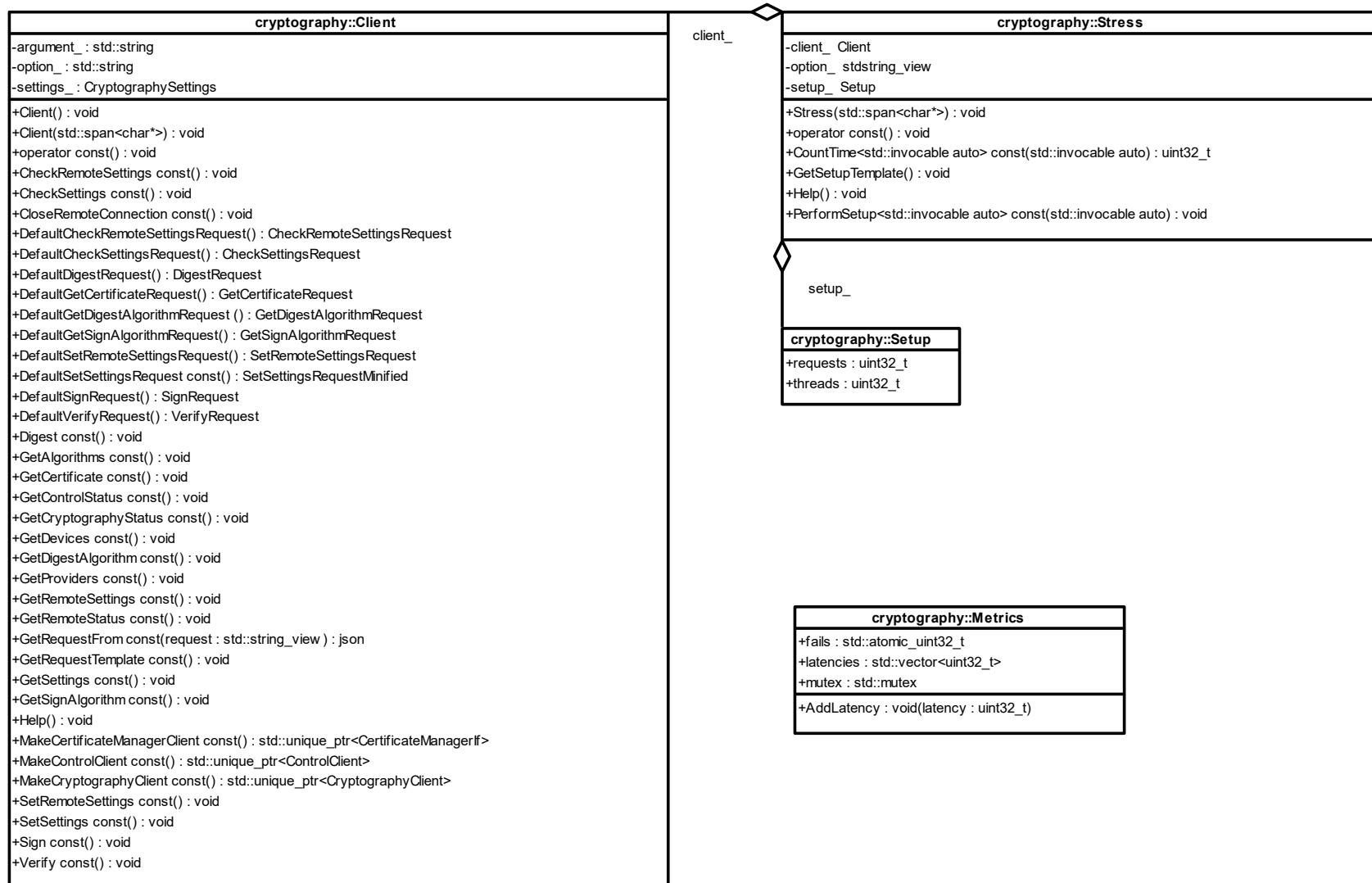


Рисунок Д.5 – Диаграмма классов утилит выполнения клиентских вызовов и стресс-тестирования

Приложение Е
(рекомендуемое)
Результаты функционального тестирования



```

1 ./cryptography develop --set-settings
2 [2025-06-16 08:09:06.020] [crypt_client] [info] [20851:20851] : SetSettings: SetSettingsRequest(
3     settings=Settings(provider=Provider(name=, description=, regime=Regime(
4     pkcs=Pkcs(), ttp_hsm=<null>, crypto_api=<null>, open_ssl=<null>, infocom=<null>)), profiles={resign: Profile(
5     digest=Algorithm(oid=1.2.643.7.1.1.2.2, uri=, parameters_oid=, description=), certificate_fields=CertInfo
6     (version=2, issuer=CN=subject, not_before=1738332794, not_after=1769868794, subject=CN=subject,
7     subject_pub_key=PubKeyInfo(algorithm=1.2.643.7.1.1.1.1, key_value=KeyValue(gost_key=GostKey(
8     alg_param=1.2.643.2.2.35.1), aki=<null>, basic_constraints=<null>, key_usage=<null>,
9     ext_key_usage=[1.3.6.1.5.5.7.3.8], alt_names=<null>, sig_alg=1.2.643.7.1.1.3.2),
10    regime_profile=RegimeProfile(pkcs=PkcsProfile(device_identity=3eed46e5, token_password=11111111),
11    ttp_hsm=<null>, crypto_api=<null>, open_ssl=<null>, infocom=<null>))), timestamp=0))

```

Рисунок Е.1 – Результат выполнения операции настройки подсистемы на стороне клиента



```
1 ./cryptography
2 [2025-06-16 13:38:56.091] [info] Запущена конечная точка: 127.0.0.1:3334
3 [2025-06-16 13:38:56.091] [info] Запущена конечная точка: 127.0.0.1:3335
4 [2025-06-16 13:38:56.092] [info] Запущена конечная точка: 127.0.0.1:3336
5 [2025-06-16 13:44:31.944] [info] Данные поставщика: Provider(
6     name=, description=, regime=Regime(
7         pkcs=Pkcs(), ttp_hsm=<null>, crypto_api=<null>, open_ssl=<null>, infocom=<null>))
8 [2025-06-16 13:44:31.944] [info] Профиль resign: pkcs
9 [2025-06-16 13:44:32.816] [info] Поиск ключевой пары по SKI
10 [2025-06-16 13:44:33.081] [info] Поиск ключевой пары по SKI Extended
11 [2025-06-16 13:44:33.114] [info] Поиск ключевой пары по открытому ключу
12 [2025-06-16 13:44:33.168] [info] Поиск ключевой пары по сертификату
13 [2025-06-16 13:44:33.202] [warning] Поиск ключевой пары без параметров
```

Рисунок Е.2 – Результат выполнения операции настройки подсистемы на стороне сервера



```
1 ./cryptography develop --sign
2 [2025-06-16 13:58:43.294] [crypt_client] [info] [29856:29856] : Sign: SignResponse(
3     signature=2916F86945E35E7D08DD9801DADE7F7657C5ACABB719EBB3A0C08E84EEBA9222
4             1806D7E698496B84F499BD8032BA0C51AC7A645A2BA05B1586C07ED4AB63F829
5     )
```

Рисунок Е.3 – Результат выполнения операции вычисления электронной подписи на стороне клиента



```

1 ./cryptography
2 [2025-06-16 14:06:02.826] [debug] Операция подписи
3 [2025-06-16 14:06:02.851] [debug] Аргументы вызова: SignRequest(
4     data=FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
5         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB4B4B4B4B4B4
6     )
7 [2025-06-16 14:10:14.922] [debug] Результат операции подписи: SignResponse(signature=
8 0000: 1f ab 11 6f 6c 3f a2 d0 4a 53 47 65 34 4d 34 ac 14 b5 58 2e 02 a8 f5 0b 33 c2 7a d2 8f 29 54 66
9 0020: 06 46 9f 0c 5e 6e aa d4 34 ea 17 0a 00 b2 8b 41 c1 e9 70 60 2e eb 5d 16 05 e3 c9 9e 2c 8e b0 d0)

```

Рисунок Е.4 – Результат выполнения операции вычисления электронной подписи на стороне сервера



```
1 ./cryptography develop --verify  
2 [2025-06-16 14:23:46.803] [debug] Verify: VerifyResponse(result: true)
```

Рисунок Е.5 – Результат выполнения операции проверки электронной подписи на стороне клиента



```
1 ./cryptography
2 [2025-06-16 14:23:46.803] [debug] Операция верификации
3 [2025-06-16 14:23:46.803] [debug] Аргументы вызова: VerifyRequest(
4     data=FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
5         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB4B4B4B4B4B4,
6     signature=
7         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
8         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
9         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
10        FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5B5B5B5B5B5B5B5B
11    )
12 [2025-06-16 14:23:46.803] [debug] Результат операции верификации: true
```

Рисунок Е.6 – Результат выполнения операции проверки электронной подписи на стороне сервера



```
1 ./cryptography develop --digest
2 [2025-06-16 14:29:28.181] [crypt_client] [info] [38015:38015] : Digest: DigestResponse(
3     digest=B4756EA97CED46B4AB7A33F1B34E6DBA5AF0AA41660DD91CD7658B6A12ABFED8
4         A4CF7EC8EEAC527995E4A81452AF071023DAC1C287CD8919B7DF28355282E4FD
5     )
```

Рисунок Е.7 – Результат выполнения операции вычисления хеш-значения на стороне клиента



```

1 ./cryptography
2 [2025-06-16 14:29:28.180] [debug] Операция хеширования
3 [2025-06-16 14:29:28.181] [debug] Аргументы вызова: DigestRequest(profile=resign, data=
4 0000: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
5 0020: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 11 11 11 11 11 11 11 11,
6 oid=1.2.643.7.1.1.2.3)
7 [2025-06-16 14:29:28.182] [debug] Результат операции хеширования: DigestResponse(
8     digest=B4756EA97CED46B4AB7A33F1B34E6DBA5AF0AA41660DD91CD7658B6A12ABFED8
9         A4CF7EC8EEAC527995E4A81452AF071023DAC1C287CD8919B7DF28355282E4FD
10     )

```

Рисунок Е.8 – Результат выполнения операции вычисления хеш-значения на стороне сервера

Приложение Ж
(рекомендуемое)
Акт о внедрении



Акционерное общество
«Информационные технологии
и коммуникационные системы»
(АО «ИнфоТеКС»)

Юридический адрес: ул. Мишина,
дом 56, стр. 2, этаж 2, пом. IX, комната 29
Москва, 127083

Почтовый адрес: а/я № 80,
улица Отрадная, дом 2Б, стр. 1
Москва, 127273

Тел. (495)737-61-92 Факс (495)737-72-78

ОГРН 1027739185066
ИНН/КПП 7710013769/771401001
<http://www.infotecs.ru>

Акционерное общество «Информационные
технологии и коммуникационные системы»
(АО «ИнфоТеКС»)

127083, Москва, улица Мишина, д. 56, стр. 2,
этаж 2, помещение IX, комната 29

.....17.06.2024..... №.....

Акт о внедрении

Настоящим подтверждаю, что Свинарев Егор Сергеевич является сотрудником компании АО «ИнфоТеКС». Решения, описанные в выпускной квалификационной работе на тему «Программные средства обеспечения унифицированного интерфейса на основе протокола PKCS#11», вошли в состав разрабатываемой технологической платформы.

Выдано Свинареву Егору Сергеевичу для предъявления по требованию.

Руководитель обособленного
подразделения в г. Пенза



М.М. Грунтович

Рисунок Г.1 – Акт о внедрении