Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Компьютерные системы и сети

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему:

**СЕРВИС ЦИФРОВОЙ ПОДПИСИ**

БГУИР КП 6-05-0612-01 105 ПЗ

Студент Головко Р.С.

Руководитель Шамына А.Ю.

Минск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc199289097)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 6](#_Toc199289098)

[1.1 Аналоги программного средства 6](#_Toc199289099)

[1.2 Постановка задачи 9](#_Toc199289100)

[2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 10](#_Toc199289101)

[2.1 Структура программы 10](#_Toc199289102)

[2.2 Проектирование функциональных требований 10](#_Toc199289103)

[2.3 Цифровая подпись 11](#_Toc199289104)

[2.4 Сервер 13](#_Toc199289105)

[2.5 Интерфейс 14](#_Toc199289106)

[2.6 Контейнеризация 15](#_Toc199289107)

[3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 17](#_Toc199289108)

[3.1 Модуль цифровой подписи 17](#_Toc199289109)

[3.2 Модуль сервера 19](#_Toc199289110)

[3.3 Модуль интерфейса 21](#_Toc199289111)

[3.4 Модуль контейнеризации 21](#_Toc199289112)

[4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 23](#_Toc199289113)

[4.1 Тестирование модуля цифровой подписи 23](#_Toc199289114)

[4.2 Тестирование модулей сервера и интерфейса 24](#_Toc199289115)

[4.3 Тестирование модуля контейнеризации 24](#_Toc199289116)

[5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 25](#_Toc199289117)

[5.1 Системные требования 25](#_Toc199289118)

[5.2 Руководство по развертыванию 25](#_Toc199289119)

[5.3 Руководство по эксплуатации 25](#_Toc199289120)

[ЗАКЛЮЧЕНИЕ 30](#_Toc199289121)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 31](#_Toc199289122)

[ПРИЛОЖЕНИЕ А 32](#_Toc199289123)

# ВВЕДЕНИЕ

В современном цифровом мире проблема обеспечения подлинности электронных документов и данных приобретает особую актуальность. Развитие интернет-технологий и переход на электронный документооборот требуют создания надежных механизмов защиты информации от подделки и несанкционированного доступа.

Целью данной курсовой работы является исследование принципов работы систем электронной цифровой подписи и разработка программного решения для реализации основных функций ЭЦП. Актуальность темы обусловлена возрастающей потребностью в простых и надежных инструментах проверки подлинности цифровых документов в различных сферах деятельности.

В работе предполагается рассмотреть следующие аспекты:

* основные принципы и алгоритмы цифровой подписи
* существующие криптографические стандарты
* методы реализации подобных систем

Особое внимание будет уделено анализу преимуществ и недостатков различных подходов к реализации механизмов электронной подписи. Также планируется исследовать возможности практического применения разрабатываемого решения.

Предполагается, что результатом работы станет программное средство, демонстрирующее базовые функции генерации и проверки электронных подписей. Разработка будет включать как теоретическую часть, посвященную анализу криптографических алгоритмов, так и практическую реализацию.

Данное исследование может представлять интерес для специалистов в области информационной безопасности, а также для разработчиков, занимающихся созданием систем защиты данных. Результаты работы могут быть использованы в качестве основы для более сложных решений в области криптографии и защиты информации.

# 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Аналоги программного средства

**1.1.1** КриптоПро CSP

КриптоПро CSP – это профессиональное программное обеспечение, предназначенное для работы с электронной подписью и криптографической защитой данных в соответствии с российскими стандартами (ГОСТ Р 34.10-2012, ГОСТ Р 34.11-2012). Решение широко используется в государственных учреждениях, банковской сфере и корпоративном документообороте, обеспечивая юридическую значимость электронных документов.

КриптоПро CSP поддерживает работу с сертифицированными средствами криптозащиты, включая токены, смарт-карты и аппаратные модули безопасности (HSM). Программа интегрируется с популярными приложениями, такими как Microsoft Office, 1С и веб-браузерами, позволяя подписывать документы без перехода в отдельное ПО.

Ключевые особенности:

* Поддержка российских и международных криптоалгоритмов (ГОСТ, RSA, ECDSA).
* Совместимость с инфраструктурой открытых ключей (PKI).
* Работа с электронной подписью в соответствии с ФЗ-63.
* Интерфейс программного средства представлен на рисунке 1.1.

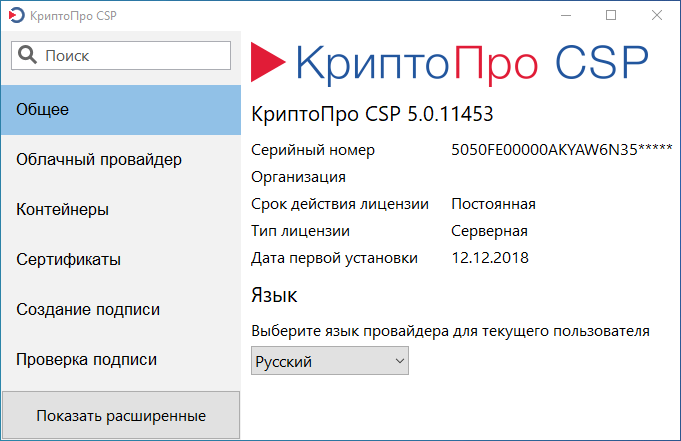


Рисунок 1.1 – Программное средство «КриптоПро CSP»

**1.1.2** GnuPG (GPG)

GnuPG (GNU Privacy Guard) — это свободное программное обеспечение с открытым исходным кодом, реализующее стандарт OpenPGP для шифрования данных и создания цифровых подписей. В отличие от коммерческих решений, GnuPG предоставляет кроссплатформенный инструментарий для криптографических операций, работающий через командную строку и графические интерфейсы (например, Kleopatra).

Основные особенности GnuPG включают:

* Поддержку популярных алгоритмов (RSA, ECC, SHA-256).
* Возможность создания и управления PGP-ключами.
* Механизмы подписи и шифрования файлов/сообщений.
* Интеграцию с почтовыми клиентами (Thunderbird, Outlook)

GnuPG широко используется в IT-среде для:

* Подписания релизов программного обеспечения.
* Шифрования электронной корреспонденции.
* Верификации git-коммитов.
* Защиты конфиденциальных данных.

Интерфейс программного средства представлен на рисунке 1.2.

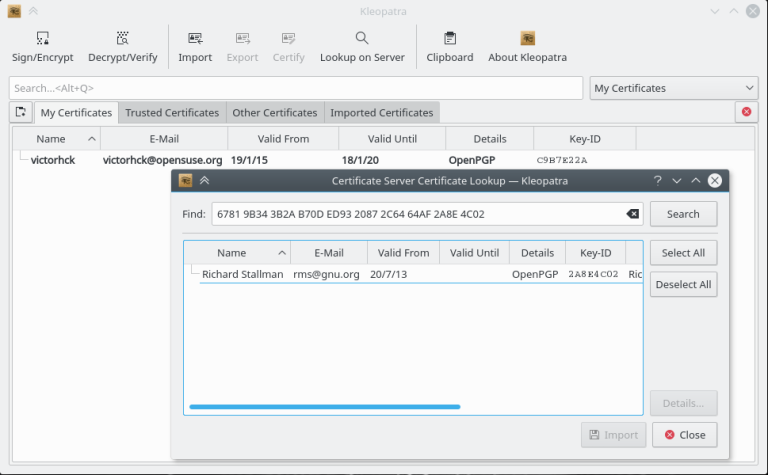


Рисунок 1.2 – Программное средство игра «GnuPG»

На рисунке представлен интерфейс программного средства Kleopatra. Оно используется в качестве графического интерфейса для GnuGP. В оригинальной версии GnuGP взаимодействие происходит через коммандную строку.

**1.1.3** DocuSign

DocuSign представляет собой облачную платформу для электронного подписания документов с юридической силой. Сервис ориентирован на бизнес-пользователей и предлагает удобный веб-интерфейс для работы с договорами, соглашениями и другими документами.

Ключевые возможности DocuSign:

* Процесс подписания документов за несколько кликов.
* Поддержка мобильных устройств.
* Интеграция с популярными CRM и ERP-системами.
* Автоматизация workflow документооборота.
* Юридическая значимость в соответствии с ESIGN Act (США) и eIDAS (ЕС).

В отличие от криптографических решений, DocuSign:

* Не требует от пользователей знаний о криптографии.
* Предоставляет готовую инфраструктуру для бизнес-процессов.
* Ориентирован на удобство, а не на технические детали подписи.

Интерфейс программного средства представлен на рисунке 1.3.

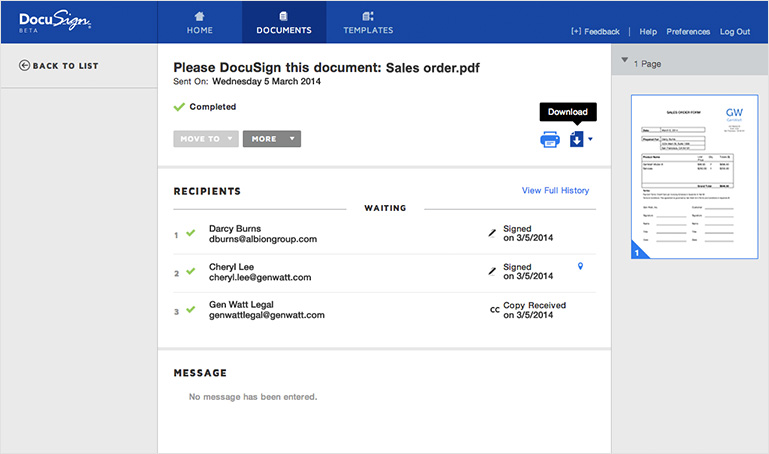


Рисунок 1.3 – Программное средство игра «DocuSign»

**1.1.4** Сравнительная характеристика аналогов

Сравнительная характеристика представлена на таблице 1.1

Таблица 1.1 – Сравнительная характеристика аналогов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | Тип | Алгоритмы | Интерфейс | Юр. сила |
| КриптоПро CSP | Коммерческое | ГОСТ, RSA | API/Плагины | ФЗ-63 |
| GnuPG | Open-source | RSA, ECC | CLI/GUI | Нет |
| DocuSign | SaaS | RSA | Веб/Мобил. | eIDAS/ESIGN |

## 1.2 Постановка задачи

Цель курсового проекта – разработка сервиса цифровой подписи на языке Golang, обеспечивающего генерацию ключей, создание и проверку электронных подписей для файлов. Сервис должен включать следующие функции:

* Генерация ключевой пары (приватный и публичный ключ) с использованием алгоритма RSA.
* Создание цифровой подписи для файлов на основе хеш-суммы SHA-256
* Верификация подписи с проверкой целостности данных.
* Простой веб-интерфейс для загрузки файлов и отображения результатов

Технические требования:

* Язык программирования: Golang.
* Среда разработки: VS Code с расширениями для Go.
* Контейнеризация: Docker (для развертывания сервиса).
* Криптография: стандартные библиотеки crypto/rsa, crypto/sha256.
* Веб-фреймворк: Gin для API и обработки HTTP-запросов.

Ожидаемые результаты:

* Работоспособный сервис с CLI и веб-интерфейсом.
* Поддержка форматов:
  + Ключи: PEM.
  + Подписи: бинарный формат с возможностью base64-кодирования.

# 2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## 2.1 Структура программы

Для реализации функций, описанных в пункте 1.2, было принято решение выделить следующие программные модули:

* Crypto
  + Генерация RSA-ключей (приватный/публичный).
  + Создание и проверка цифровой подписи с использованием SHA-256 .
  + Кодирование ключей в PEM-формат.
* Network
  + Обработка HTTP-запросов (REST API).
  + Взаимодействие с клиентом через веб-интерфейс.
* Web
  + Веб-интерфейс для загрузки файлов и отображения результатов.

## 2.2 Проектирование функциональных требований

Для обеспечения работоспособности сервиса цифровой подписи были сформулированы следующие ключевые функциональные требования:

1. Требования к генерации ключей

* Генерация пары RSA-ключей (приватный/публичный) длиной 2048 бит.
* Автоматическое преобразование ключей в PEM-формат.
* Возможность скачивания ключей в виде ZIP-архива (certificate.pem + private\_key.pem).

2. Требования к созданию подписи

* Корректное вычисление SHA-256 хеша для файлов любого типа.
* Создание цифровой подписи с использованием RSA-PKCS#1 v1.5.
* Поддержка файлов размером до 1 ГБ (с потоковой обработкой).

3. Требования к проверке подписи

* Верификация подписи с проверкой:
  + Соответствия хеша исходного файла.
  + Корректности структуры подписи.
  + Валидности сертификата.

4. Требования к веб-интерфейсу

* Интуитивный интерфейс для:
  + Загрузки файлов.
  + Отображения статуса операций.
  + Вывода результатов проверки.

5. Требования безопасности

* Запрет на хранение приватных ключей после завершения сеанса.
* Валидация входных данных (размер, формат).

## 2.3 Цифровая подпись

**2.3.1** Криптография с открытым ключом

Цифровая подпись представляет собой криптографический механизм, обеспечивающий три ключевых свойства электронных документов: аутентификацию источника данных, гарантию целостности информации и невозможность отказа от авторства (неотрекаемость). В основе системы лежит асимметричная криптография, использующая пару математически связанных ключей: открытый (публичный) и закрытый (приватный).

Принцип работы:

* Закрытый ключ используется для создания подписи и хранится в секрете
* Открытый ключ применяется для проверки подписи и свободно распространяется
* Любое изменение подписанных данных делает подпись недействительной

**2.3.2** Выбор алгоритма RSA

Для реализации был выбран алгоритм RSA как наиболее изученный стандарт асимметричной криптографии. Его безопасность основана на вычислительной сложности факторизации больших целых чисел.

Математические основы:

1. Генерация ключей:
   * Выбираются два больших простых числа p и q
   * Вычисляется модуль n = p × q
   * Определяется функция Эйлера φ(n) = (p-1)\*(q-1)
   * Выбирается открытая экспонента e
   * Вычисляется закрытая экспонента d ≡ e⁻¹ mod φ(n)
2. Операции:
   * Подпись: s ≡ mᵈ mod n
   * Проверка: m ≡ sᵉ mod n

Преимущества выбора:

* Широкая поддержка в стандартных библиотеках
* Понятная математическая модель
* Достаточная стойкость при длине ключа 2048+ бит

**2.3.3** Хеш-функции и их роль

Перед подписанием данные обязательно хешируются. В системе используется SHA-256 как часть стандарта SHA-2.

Характеристики SHA-256:

* Размер выхода: 256 бит (32 байта)
* Стойкость к коллизиям
* Вычислительная эффективность

Процесс хеширования:

1. Разбиение входных данных на блоки по 512 бит
2. Последовательная обработка блоков
3. Применение 64 раундов преобразований
4. Формирование итогового значения хеша

**2.3.4** Стандарты формирования подписи

В системе реализована схема RSASSA-PKCS1-v1.5, описанная в RFC 8017. Данный стандарт предусматривает:

Структура блока подписи представлена на рисунке 2.3.1.

0x00 || 0x01 || PS || 0x00 || T

Рисунок 2.3.1 – Структура блока подписи

Условные обозначения:

* PS - байты заполнения (0xFF)
* T - DER-кодированный хеш

DER-кодирование включает:

* Версию (0)
* Модуль (n)
* Публичную экспоненту (e)
* Приватную экспоненту (d)
* Простые числа (p, q)
* CRT-коэффициенты

Критика и ограничения:

* Уязвимость к атакам при неправильной реализации
* Моральное устаревание в пользу RSASSA-PSS
* Сохранение совместимости с существующими системами

**2.3.5** Форматы хранения ключей

В системе использован формат PEM (Privacy-Enhanced Mail) для хранения ключей:

Структура PEM-файла представлена на рисунке 2.3.2:

-----BEGIN RSA PRIVATE KEY-----

Base64-кодированные данные

-----END RSA PRIVATE KEY-----

Рисунок 2.3.2 – Структура PEM-файла

**2.3.6** Альтернативные алгоритмы и перспективы

Хотя RSA был выбран для данной реализации, современные системы часто используют:

1. ECDSA (на эллиптических кривых):
   1. Меньшая длина ключей при сопоставимой стойкости
   2. Высокая вычислительная эффективность
   3. Сложность реализации
2. Алгоритмы на основе решёток:
   1. Квантово-устойчивые решения
   2. Активная область исследований

## 2.4 Сервер

**2.4.1** Работа сервера

Серверный модуль представляет собой REST API, обеспечивающее криптографические операции через HTTP-интерфейс. Ключевые требования:

* Масштабируемость: Поддержка 100+ одновременных запросов
* Простота интеграции: Четко определенные JSON-форматы запросов/ответов

Схема работы сервера представлена на рисунке 2.4.1

Клиент → HTTP-запрос → Маршрутизатор → Обработчик → Криптомодуль → Формирование ответа

Рисунок 2.4.1 – Схема работы сервера

**2.4.2** Выбор Gin Framework

Аргументы в пользу Gin:

* Производительность: до 40k RPS благодаря оптимизированному роутеру
* Минимализм: 10 ключевых методов API
* Middleware: Готовые решения для:
  + CORS
  + Rate-limiting
  + Gzip-сжатия

Сравнение с альтернативами представлено на таблице 2.4.1

Таблица 2.4.1 – Сравнение framework-ов

|  |  |  |  |
| --- | --- | --- | --- |
| Framework | Использование RAM | JSON скорость | Документация |
| Gin | 4MB | 1x | Хорошая |
| Echo | 5MB | 0.9x | Средняя |
| Fiber | 3MB | 1.2x | Ниже среднего |

**2.4.3** Архитектура API

Эндпоинты:

1. GET /api/generate

Сделан для генерации сертификата и приватного ключа и отправки их клиенту.

* Ответ: ZIP-архив с:
  + certificate.pem (X.509)
  + private\_key.pem (PKCS#1)

1. POST /api/sign

Сделан для подписания файла любого типа.

* Параметры:
  + file (бинарные данные)
  + key (PEM-кодированный)
* Валидация:
  + Макс. размер файла: 1GB
  + MIME-типы: application/octet-stream, text/\*
* Ответ:
  + file.sig(файл цифровой подписи)

1. POST /api/verify

Сделан для проверки подписи по сертификату.

* Параметры:
  + file (бинарные данные)
  + signature (DER-кодированная)
  + sertificate (PEM-кодированный)
* Ответ:
  + Сообщение json с ответом

## 2.5 Интерфейс

**2.5.1** Архитектура

Фронтенд реализован как SPA (Single Page Application) с тремя ключевыми страницами:

* Генерация ключевой пары
* Создание цифровой подписи
* Верификация подписи

Принцип работы показан на рисунке 2.5.1.

Пользователь → Взаимодействие с UI → Fetch API →

→ Сервер → Обработка ответа → Отображение результата

Рисунок 2.5.1 – Схема работы фронтенда

**2.5.2** Выбор технологий

Минималистичный стек:

* Чистый HTML/CSS без фреймворков
* Ванильный JavaScript
* Fetch API для коммуникации с сервером

Преимущества:

* Нет зависимостей от сторонних библиотек
* Быстрая загрузка (<100KB всех ресурсов)
* Полный контроль над поведением

**2.5.3** Интерфейс

1. Главная страница (index.html):

* Навигационные кнопки с переходами
* Минималистичный дизайн

2. Генерация ключей (generate.html):

* Индикатор загрузки (CSS-анимация)
* Автоматическое скачивание ZIP-архива
* Резервная ссылка для manual download

3. Подпись файлов (sign.html):

* Форма с:
  + Поле выбора файла
  + Поле для приватного ключа (PEM)
* Результат с:
  + Размером подписи
  + Автоскачиванием .sig файла

4. Верификация (verify.html):

* Тройная загрузка:
  + Исходный файл
  + Подпись
  + Сертификат
* Визуализация результата (✓/✗)

**2.5.4** Безопасность

Реализованные меры:

* Защита данных:
  + Ключи не сохраняются в localStorage
  + Очистка FormData после операций
* Валидация:
  + Проверка наличия файлов перед отправкой

## 2.6 Контейнеризация

**2.6.1** Обоснование выбора Docker

Docker обеспечивает:

* Изолированность – все зависимости содержатся внутри контейнера
* Переносимость – идентичная работа на любых системах (dev/prod)
* Масштабируемость – быстрое развертывание дополнительных экземпляров

Альтернативы и их недостатки:

* Ручная установка: требует настройки среды на каждом сервере
* Виртуальные машины: избыточное потребление ресурсов (~ГБ памяти на экземпляр)
* Kubernetes: излишняя сложность для данного проекта

**2.6.2** Dockerfile

Для уменьшения размера результирующего образа я решил использовать двухэтапную сборку. Пример кода на рисунке 2.6.1

FROM golang:1.24-alpine AS builder # Этап сборки

...

FROM alpine:latest # Этап выполнения

Рисунок 2.6.1 – Пример двухэтапной сборки

Ключевые особенности:

1. Базовый образ:

* golang:1.24-alpine – минимальный образ с Go (включает компилятор)
* alpine:latest – итоговый образ (~5MB)

1. Оптимизации:

* Отдельная загрузка зависимостей (go mod download)

Для упрощения развертывания можно использовать простейший файл compose.yaml. На рисунке 2.6.2 представлено простейшее содержание compose файла.

services:

app:

build: .

ports:

- "8080:8080"

Рисунок 2.6.2 – Простейший compose.yaml

# 3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

Для разработки сервиса цифровой подписи я выбрал язык Golang и сетевой framework Gin. Мой выбор обусловлен высокой производительностью Go при обработке криптографических операций, простотой создания надежных сетевых приложений благодаря строгой типизации и встроенным механизмам конкурентности, а также минималистичному и эффективному роутингу, который предоставляет Gin. Использование Docker для контейнеризации позволило обеспечить стабильную работу сервиса в различных окружениях и упростить процесс развертывания. Такой технологический стек обеспечил оптимальное сочетание скорости разработки, надежности и простоты дальнейшего сопровождения проекта.

## 3.1 Модуль цифровой подписи

**3.1.1** Взаимодействие с модулем

Для работы с цифровой подписью был разработан интерфейс Signer, код структуры интерфейса представлен ниже

type Signer interface {

    GenCertAndKey() (\*x509.Certificate, \*rsa.PrivateKey, error)

    Sign(data []byte, key \*rsa.PrivateKey) ([]byte, error)

    Verify(data []byte, signature []byte, cert \*x509.Certificate) error

    CertMarshal(cert \*x509.Certificate) []byte

    KeyMarshal(key \*rsa.PrivateKey) []byte

    CertUnmarshal(data []byte) (\*x509.Certificate, error)

    KeyUnmarshal(data []byte) (\*rsa.PrivateKey, error)

    IsCertValid(cert \*x509.Certificate) bool

}

Методы интерфейса и их функции:

* GenCertAndKey – Генерирация сертификата, который используется в качестве открытого ключа, и приватного ключа.
* Sign – Подписание документа.
* Verify – Проверка подписи.
* Методы оканчивающиеся на Marshal – сериализация.
* Методы оканчивающиеся на Unmarshal – десериализация.
* IsCertValid – Проверка срока действия сертификата.

**3.1.2** Внутренняя реализация

Объект реализующий вышеприведенный интерфейс имеет тип RSASigner, ниже приведен код данного типа.

type RSASigner struct {

    serialNumber \*big.Int

    organization []string

    commonName   string

    keyLength    int

    duration     time.Duration

}

Описание полей RSASigner:

* serialNumber – серийный номер сертификата
* organization – название организации, используется для сертификата
* commonName – название организации обыденное, используется для сертификата
* keyLength – длина ключа
* duration – время действия сертификата

Все содержимое полей заполняется при создании объекта, данные для полей берутся из файла с настройками.

**3.1.3** Подпись файла

Для подписи файла используется метод Sign, в реализации метода используются библиотеки crypto/rsa и crypto/sha256. Код данного метода приведен ниже.

func (s \*RSASigner) Sign(data []byte, key \*rsa.PrivateKey) ([]byte, error) {

    if len(data) == 0 {

        return nil, &CryptoError{

            Type:    SignErr,

            Err:     errors.New("empty data"),

            Content: "data to sign cannot be empty",

        }

    }

    hashed := sha256.Sum256(data)

    signature, err := rsa.SignPKCS1v15(rand.Reader, key, crypto.SHA256, hashed[:])

    if err != nil {

        return nil, &CryptoError{

            Type:    SignErr,

            Err:     err,

            Content: "failed to sign data",

        }

    }

    return signature, nil

}

**3.1.4** Проверка подписи

Для проверки подписи файла используется метод Verify, в реализации метода используются библиотеки crypto/rsa и crypto/sha256. Код данного метода приведен ниже.

func (s \*RSASigner) Verify(data []byte, signature []byte, cert \*x509.Certificate) error {

    if len(data) == 0 || len(signature) == 0 {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("empty data or signature"),

            Content: "data and signature cannot be empty",

        }

    }

    if !s.IsCertValid(cert) {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("certificate expired or not yet valid"),

            Content: "certificate validation failed",

        }

    }

    hashed := sha256.Sum256(data)

    pubKey, ok := cert.PublicKey.(\*rsa.PublicKey)

    if !ok {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("invalid public key type"),

            Content: "certificate does not contain RSA public key",

        }

    }

    if err := rsa.VerifyPKCS1v15(pubKey, crypto.SHA256, hashed[:], signature); err != nil {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     err,

            Content: "signature verification failed",

        }

    }

    return nil

}

Функция IsSertValid проверяет срок действия сертификата, код данной функции представлен ниже.

func (s \*RSASigner) IsCertValid(cert \*x509.Certificate) bool {

    now := time.Now()

    return now.After(cert.NotBefore) && now.Before(cert.NotAfter)

}

Полный код модуля Crypto представлен в приложении А.

## 3.2 Модуль сервера

**3.2.1** Взаимодействие с модулем

Для взаимодействия с модулем используется структура типа Router, код структуры представлен ниже.

type Router struct {

    engine \*gin.Engine

    signer crypto.Signer

}

Описание полей:

* engine – Мультиплексор фреймворка Gin
* singer – Интерфейс для работы с цифровой подписью

**3.2.2** Обработчик маршрута подписи

В обработчике маршрута для подписи файла реализована потоковая обработка файла, код метода приведен ниже.

func (r \*Router) signHandler(c \*gin.Context) {

    dataFile, err := c.FormFile("data")

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Необходимо загрузить файл данных",

            "details": err.Error(),

        })

        return

    }

    file, err := dataFile.Open()

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка чтения файла данных",

            "details": err.Error(),

        })

        return

    }

    defer file.Close()

    hasher := sha256.New()

    if \_, err := io.Copy(hasher, file); err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка вычисления хеша",

            "details": err.Error(),

        })

        return

    }

    hash := hasher.Sum(nil)

    keyContent := c.PostForm("key")

    key, err := r.signer.KeyUnmarshal([]byte(keyContent))

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Неверный формат приватного ключа",

            "details": err.Error(),

        })

        return

    }

    signature, err := rsa.SignPKCS1v15(rand.Reader, key, crypto.SHA256, hash)

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка создания подписи",

            "details": err.Error(),

        })

        return

    }

    c.JSON(http.StatusOK, gin.H{

        "signature": base64.StdEncoding.EncodeToString(signature),

    })

}

Полный код модуля Network представлен в приложении А.

## 3.3 Модуль интерфейса

Для взаимодействия с приложением сделаны 4 страницы, код разметки главной страницы на HTML представлен ниже.

<!DOCTYPE html>

<html lang="ru">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Цифровая подпись</title>

    <link rel="stylesheet" href="/static/css/style.css">

</head>

<body>

    <div class="container">

        <h1>Сервис цифровой подписи</h1>

        <div class="menu">

            <a href="/generate" class="btn">Генерация сертификата</a>

            <a href="/sign" class="btn">Подпись данных</a>

            <a href="/verify" class="btn">Проверка подписи</a>

        </div>

    </div>

</body>

</html>

## 3.4 Модуль контейнеризации

При создании контейнера использовалась двухэтапная сборка, и легковесные образы alpine, что уменьшило размер контейнера по сравнению со стандартными образами или одноэтапной сборкой, содержимое докер файла представлено ниже.

FROM golang:1.24-alpine AS builder

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN GOOS=linux go build -o /go-digisign ./cmd/app/main.go

FROM alpine:latest

WORKDIR /app

COPY --from=builder /go-digisign /app/go-digisign

COPY --from=builder /app/static ./static

EXPOSE 8080

CMD ["/app/go-digisign"]

Для compose файла выбран пример с простейшим содержимым, оно представлено ниже.

services:

  app:

    build: .

    ports:

      - "8080:8080"

# 4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## 4.1 Тестирование модуля цифровой подписи

Для тестирования модуля цифровой подписи мною были написаны тесты, код одного из них представлен ниже.

func TestSigner(t \*testing.T) {

    signature, err := signer.Sign(data, key)

    if err != nil {

        t.Errorf("%v,\n", err)

    }

    log.Println("create a digital sign: success")

    if err := signer.Verify(data, signature, cert); err != nil {

        t.Errorf("%v,\n", err)

    }

    log.Println("verify a digital sign: success")

    keyPEM := signer.KeyMarshal(key)

    log.Println("key to bytes:\n", string(keyPEM))

    \_, err = signer.KeyUnmarshal(keyPEM)

    if err != nil {

        t.Errorf("%v,\n", err)

    }

    log.Println("bytes to key: success")

    certPEM := signer.CertMarshal(cert)

    log.Println("cert to bytes:\n", string(certPEM))

    \_, err = signer.CertUnmarshal(certPEM)

    if err != nil {

        t.Errorf("%v,\n", err)

    }

    log.Println("bytes to key: success")

}

При запуске тестов получен результат:

=== RUN TestSigner

2025/05/27 23:20:20 create a digital sign: success

2025/05/27 23:20:20 verify a digital sign: success

2025/05/27 23:20:20 key to bytes:

-----BEGIN RSA PRIVATE KEY-----

MIIEowIBAAKCAQEAvUTcnR…

-----END RSA PRIVATE KEY-----

2025/05/27 23:20:20 bytes to key: success

2025/05/27 23:20:20 cert to bytes:

-----BEGIN CERTIFICATE-----

MIIDHDCCAgSgAwIBAgIBAT…

-----END CERTIFICATE-----

2025/05/27 23:20:20 bytes to key: success

--- PASS: TestSigner (0.00s)

=== RUN TestEndToEnd

--- PASS: TestEndToEnd (0.00s)

PASS

ok github.com/koan6gi/go-digisign/internal/crypto 0.567s

## 4.2 Тестирование модулей сервера и интерфейса

При переписывании обработчика для получения сертификата и ключа была допущена ошибка в его инициализации, он был настроен на прием только POST-запросов:

r.engine.POST("/api/generate", r.generateHandler)

Однако с фронтенда приходил GET-запрос, после изменения инициализации, программа заработала корректно.

r.engine.GET("/api/generate", r.generateHandler)

## 4.3 Тестирование модуля контейнеризации

При тестировании модуля контейнеризации ошибки не были обнаружены. Проект корректно развертывается и завершается.

# 5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## 5.1 Системные требования

Для работы данной программы компьютер пользователя должен соответствовать следующим требованиям:

– наличие Docker;

– наличие любого браузера;

– 20 Мб свободного дискового пространства;

## 5.2 Руководство по развертыванию

Для запуска приложения предварительно нужно запустить docker engine, далее в папке с проектом необходимо открыть консоль и ввести и запустить команду:

docker compose up

Доступ к приложению будет по адресу:

http://localhost:8080

Для отключения приложения нужно нажать комбинацию клавиш Ctrl+C или в консоли в папке с проектом ввести и запустить комманду:

docker compose down

## 5.3 Руководство по эксплуатации

После открытия приложения в браузере будет отображено начальное меню. В главном окне будут отображены три кнопки: Генерация сертификата, Подпись данных, Проверка подписи. Внешний вид данной страницы изображен на рисунке 5.1.

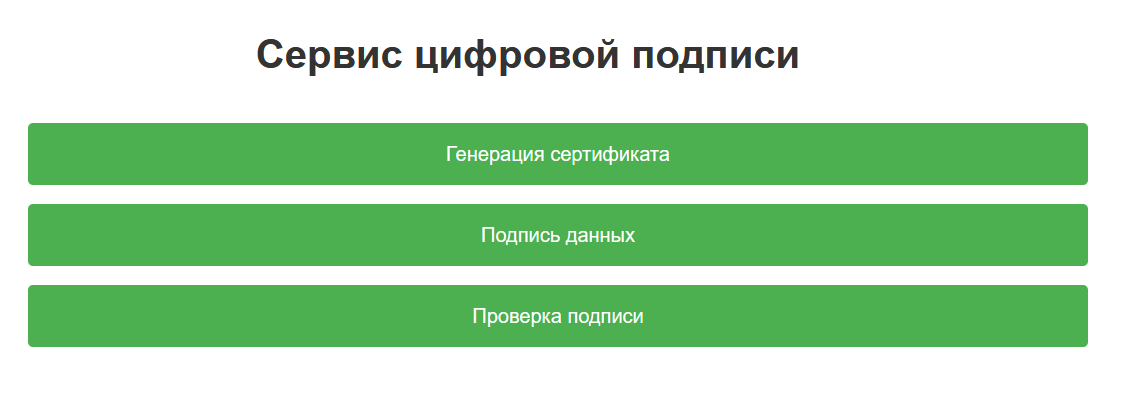


Рисунок 5.1 – Изображение главного меню.

При нажатии на кнопку Генерация сертификата вы будете перемещены на страницу для генерации сертификата и приватного ключа, внешний вид данной страницы отображен на рисунке 5.2.

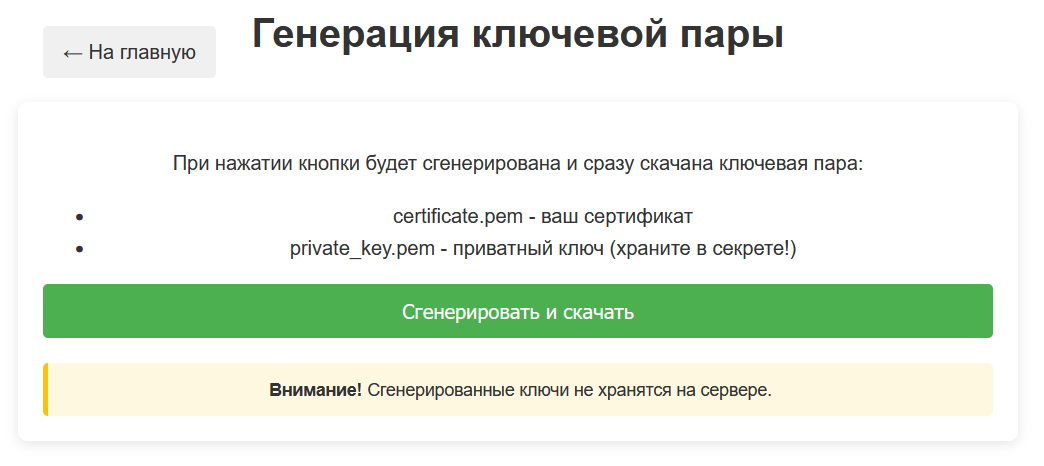


Рисунок 5.2 – Страница Генерация ключевой пары

После нажатия на кнопку Сгенерировать и скачать вам будет закачан архив credentials.zip с приватным ключом и сертификатом. Процесс загрузки на рисунке 5.3

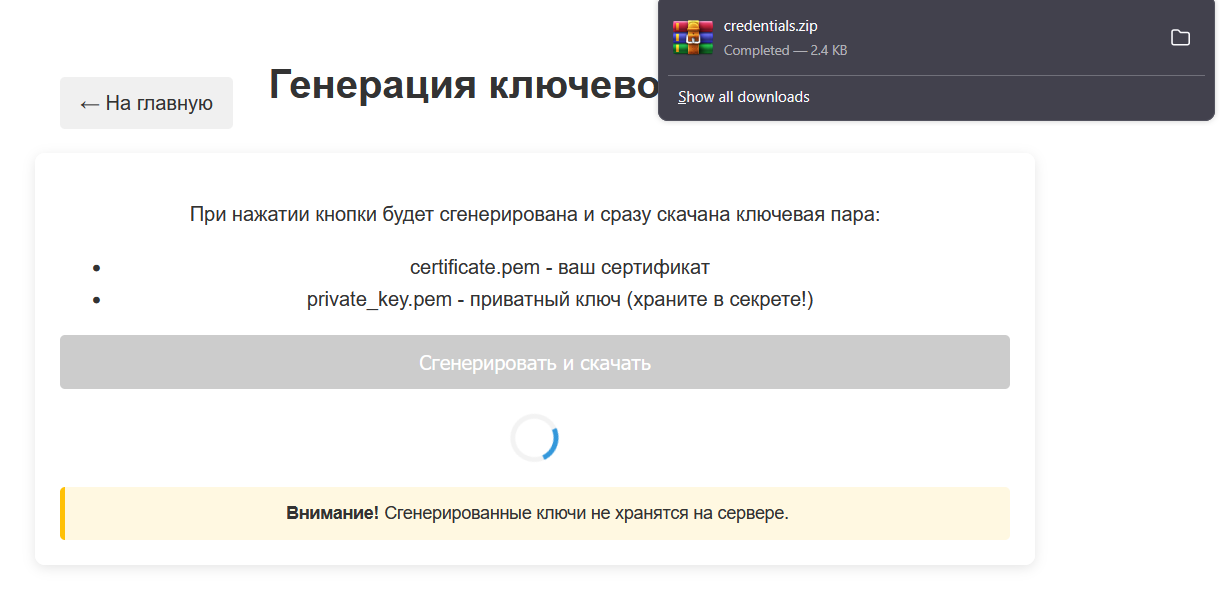


Рисунок 5.3 – Процесс загрузки ключевой пары

Для возврата к главной странице необходимо нажать на кнопку На главную.

При нажатии на кнопку Подпись данных на главной странице вы будете перемещены на страницу подписи данных. Изображение страницы подписи на рисунке 5.4.

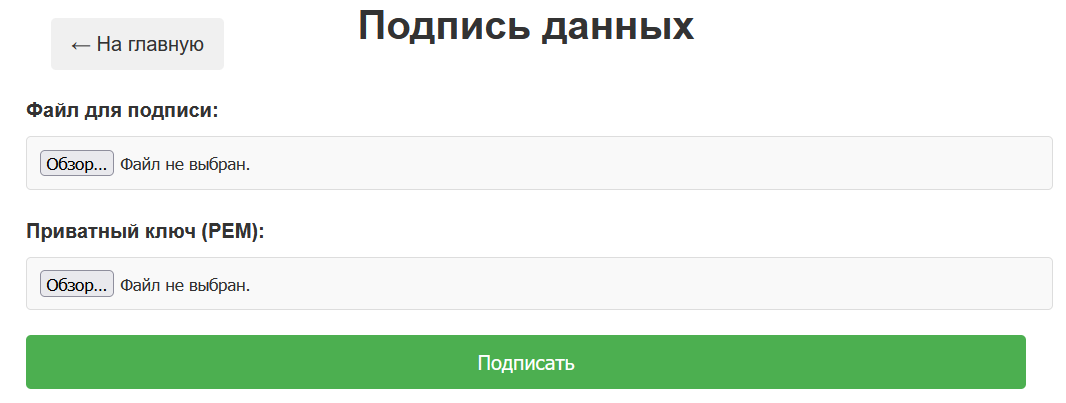


Рисунок 5.4 – Подпись данных

Для подписи первоначально необходимо выбрать файл, который вы хотите подписать, далее необходимо выбрать файл приватного ключа, который получен на странице Генерация ключевой пары. Результат подписания на рисунке 5.5.

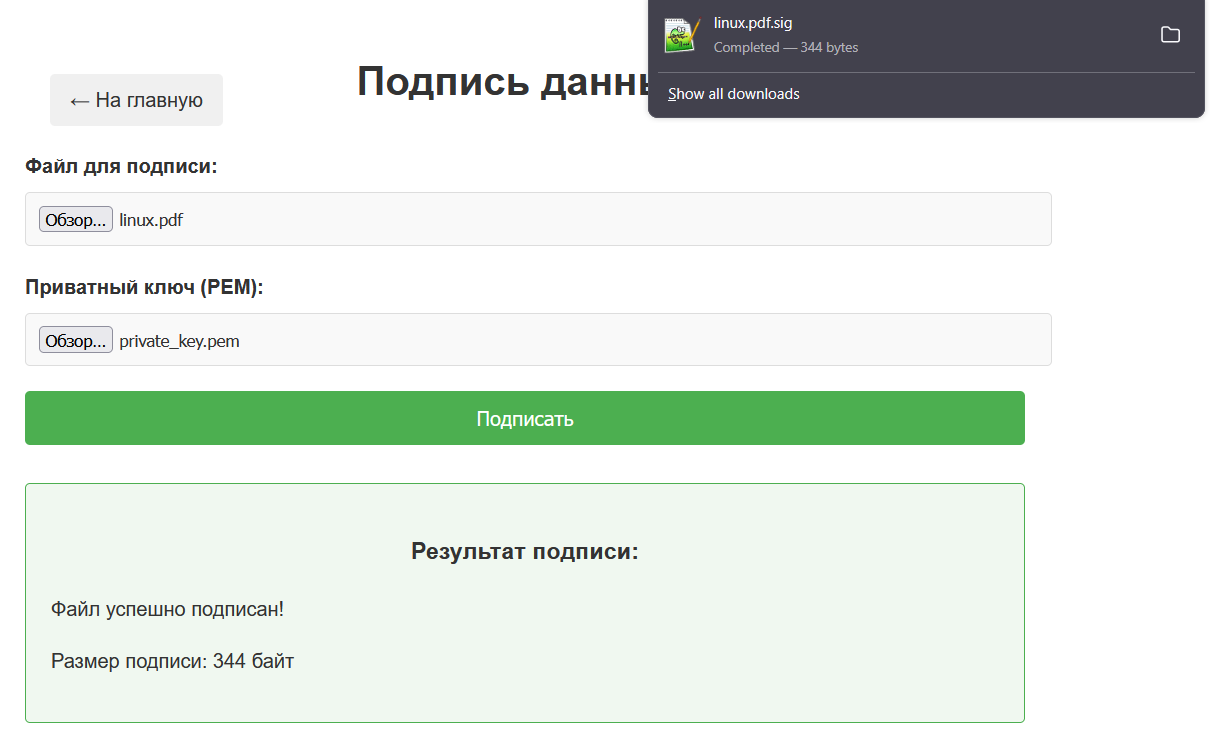


Рисунок 5.5 – Подпись данных результат

В результате появится окно с результатом подписания и к вам на компьютер скачается файл подписи с расширением .sig.

Для возврата на главную страницу нажмите на кнопку На главную.

При нажатии на кнопку Проверка подписи на главной странице вы перейдете на страницу проверки подписи. Содержимое страницы на рисунке 5.6.

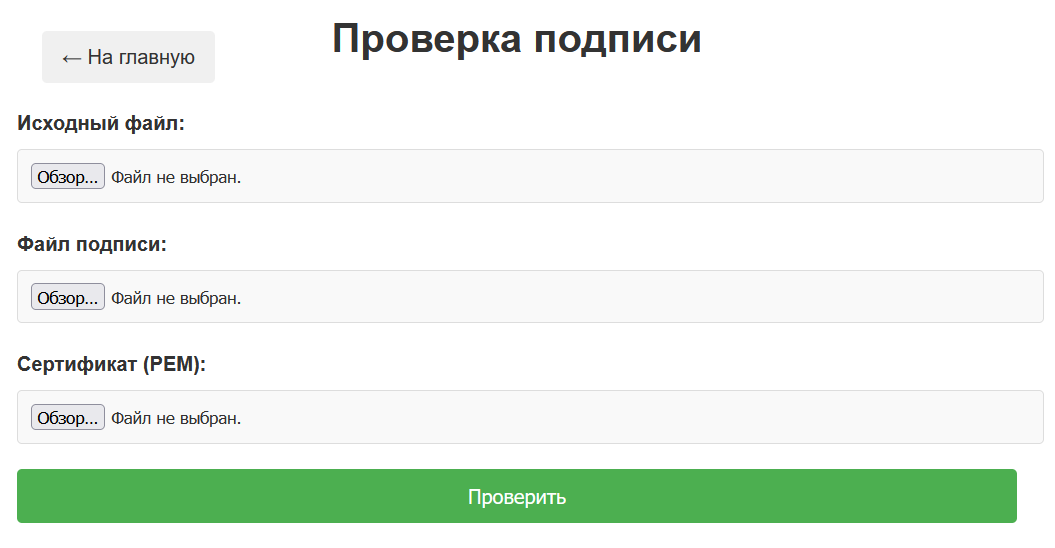


Рисунок 5.6 – Проверка подписи

Для проверки подписи необходимо выбрать исходный файл, файл подписи, полученный на странице подпись данных, и файл сертификата, полученный на странице. Результат проверки на рисунке 5.7.

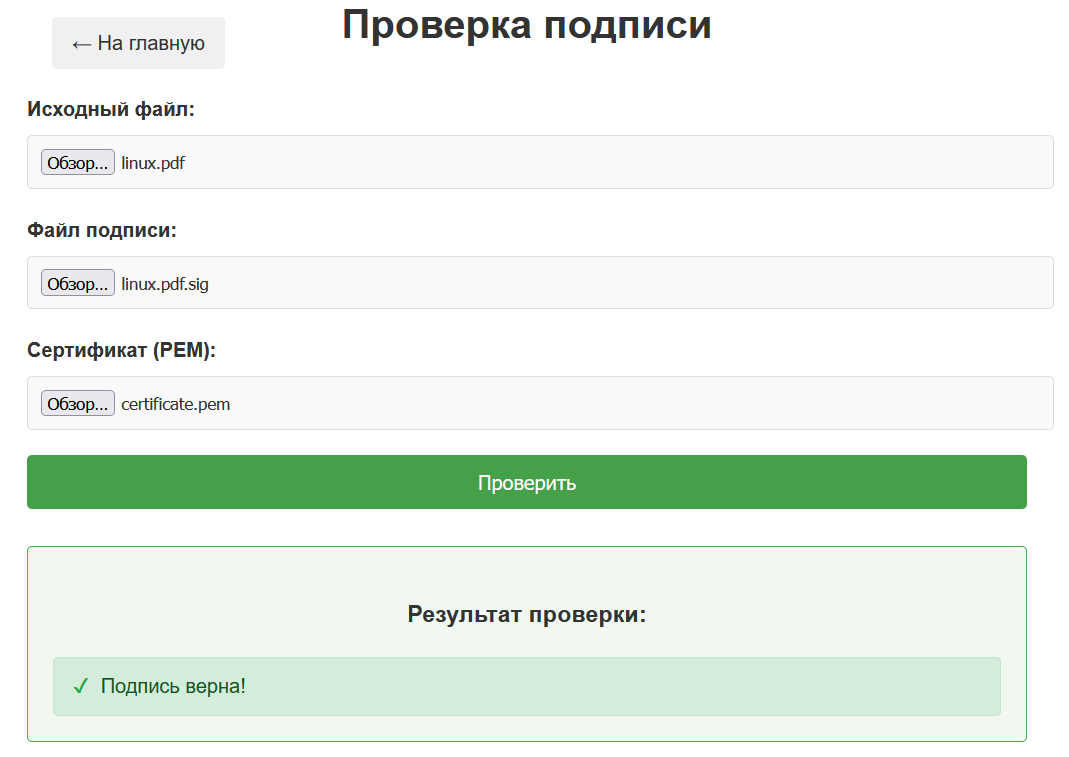


Рисунок 5.7 – Проверка подписи результат

В результате проверки появится окно с ответом подпись верна (рисунок 5.8) или проверка подписи не пройдена (рисунок 5.9).

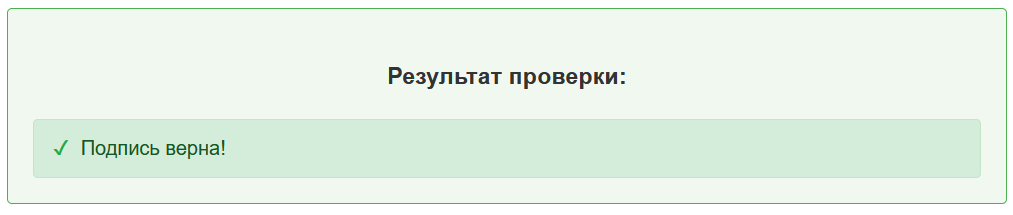


Рисунок 5.8 – Подпись верна.

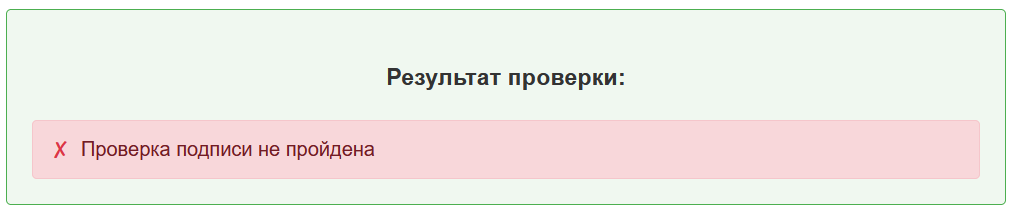


Рисунок 5.9 – Проверка подписи не пройдена.

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта было разработано программное обеспечение для создания и проверки электронной цифровой подписи (ЭЦП) на основе алгоритма RSA. Сервис предоставляет пользователю удобный веб-интерфейс для генерации ключевой пары, подписания файлов и их верификации.

Проект демонстрирует базовые принципы работы ЭЦП и может быть использован в образовательных целях для изучения криптографии. Также он служит основой для более сложных систем, таких как:

* интеграция с облачными хранилищами;
* поддержка других алгоритмов (ECDSA, ГОСТ Р 34.10-2012).

Разработанный сервис успешно выполняет поставленные задачи и подтверждает работоспособность выбранных технологий. Несмотря на учебный характер проекта, он закладывает фундамент для дальнейшего развития в полноценное решение для безопасного документооборота.

Перспективы**:**

* Добавление PGP-подписи
* Поддержка аппаратных токенов
* Развитие системы как SaaS-платформы

Работа над проектом позволила углубить знания в криптографии, веб-разработке и DevOps, что будет полезно для будущих профессиональных задач в области информационной безопасности.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1]. *Нильс Фергюсон, Брюс Шнайер.* Практическая криптография = Practical Cryptography: Designing and Implementing Secure Cryptographic Systems. — М. : Диалектика, 2004. — 432 с.

[2]. *Шнайер Б.* Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си = Applied Cryptography. Protocols, Algorithms and Source Code in C. — М.: Триумф, 2002. — 816 с.

[3]. *Rivest R., Shamir A., Adleman L.* A method for obtaining digital signatures and public-key cryptosystems — 15 c.

[4]. Gin Web Framework [Электронный ресурс]. – Режим доступа: https://gin-gonic.com.

[5]. Gin [Электронный ресурс]. – Режим доступа: https://pkg.go.dev/github.com/gin-gonic/gin.

# ПРИЛОЖЕНИЕ А

**Исходный код**

**Network**

package crypto

import (

    "crypto"

    "crypto/rand"

    "crypto/rsa"

    "crypto/sha256"

    "crypto/x509"

    "crypto/x509/pkix"

    "encoding/pem"

    "errors"

    "math/big"

    "time"

    "github.com/koan6gi/go-digisign/internal/config"

)

type Signer interface {

    GenCertAndKey() (\*x509.Certificate, \*rsa.PrivateKey, error)

    Sign(data []byte, key \*rsa.PrivateKey) ([]byte, error)

    Verify(data []byte, signature []byte, cert \*x509.Certificate) error

    CertMarshal(cert \*x509.Certificate) []byte

    KeyMarshal(key \*rsa.PrivateKey) []byte

    CertUnmarshal(data []byte) (\*x509.Certificate, error)

    KeyUnmarshal(data []byte) (\*rsa.PrivateKey, error)

    IsCertValid(cert \*x509.Certificate) bool

}

type CryptoError struct {

    Type    int

    Err     error

    Content string

}

func (e \*CryptoError) Error() string { return e.Content }

func (e \*CryptoError) Unwrap() error { return e.Err }

const (

    GenKeyErr = iota

    GenCertErr

    SignErr

    VerifyErr

    DataErr

)

type RSASigner struct {

    serialNumber \*big.Int

    organization []string

    commonName   string

    keyLength    int

    duration     time.Duration

}

func NewRSASigner() \*RSASigner {

    keyLength := config.KeyLength

    duration := config.CertDuration

    return &RSASigner{

        serialNumber: big.NewInt(1),

        organization: []string{config.Organization},

        commonName:   config.CommonName,

        keyLength:    keyLength,

        duration:     duration,

    }

}

func (s \*RSASigner) GenCertAndKey() (\*x509.Certificate, \*rsa.PrivateKey, error) {

    privateKey, err := rsa.GenerateKey(rand.Reader, s.keyLength)

    if err != nil {

        return nil, nil, &CryptoError{

            Type:    GenKeyErr,

            Err:     err,

            Content: "failed to generate RSA private key",

        }

    }

    certTemplate := &x509.Certificate{

        SerialNumber: s.serialNumber,

        Subject: pkix.Name{

            Organization: s.organization,

            CommonName:   s.commonName,

        },

        NotBefore: time.Now(),

        NotAfter:  time.Now().Add(s.duration),

        KeyUsage:  x509.KeyUsageDigitalSignature,

        ExtKeyUsage: []x509.ExtKeyUsage{

            x509.ExtKeyUsageClientAuth,

            x509.ExtKeyUsageServerAuth,

        },

        BasicConstraintsValid: true,

        IsCA:                  false,

    }

    certDER, err := x509.CreateCertificate(rand.Reader, certTemplate, certTemplate, &privateKey.PublicKey, privateKey)

    if err != nil {

        return nil, nil, &CryptoError{

            Type:    GenCertErr,

            Err:     err,

            Content: "failed to create x509 certificate",

        }

    }

    cert, err := x509.ParseCertificate(certDER)

    if err != nil {

        return nil, nil, &CryptoError{

            Type:    GenCertErr,

            Err:     err,

            Content: "failed to parse DER certificate",

        }

    }

    s.serialNumber.Add(s.serialNumber, big.NewInt(1))

    return cert, privateKey, nil

}

func (s \*RSASigner) Sign(data []byte, key \*rsa.PrivateKey) ([]byte, error) {

    if len(data) == 0 {

        return nil, &CryptoError{

            Type:    SignErr,

            Err:     errors.New("empty data"),

            Content: "data to sign cannot be empty",

        }

    }

    hashed := sha256.Sum256(data)

    signature, err := rsa.SignPKCS1v15(rand.Reader, key, crypto.SHA256, hashed[:])

    if err != nil {

        return nil, &CryptoError{

            Type:    SignErr,

            Err:     err,

            Content: "failed to sign data",

        }

    }

    return signature, nil

}

func (s \*RSASigner) Verify(data []byte, signature []byte, cert \*x509.Certificate) error {

    if len(data) == 0 || len(signature) == 0 {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("empty data or signature"),

            Content: "data and signature cannot be empty",

        }

    }

    if !s.IsCertValid(cert) {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("certificate expired or not yet valid"),

            Content: "certificate validation failed",

        }

    }

    hashed := sha256.Sum256(data)

    pubKey, ok := cert.PublicKey.(\*rsa.PublicKey)

    if !ok {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     errors.New("invalid public key type"),

            Content: "certificate does not contain RSA public key",

        }

    }

    if err := rsa.VerifyPKCS1v15(pubKey, crypto.SHA256, hashed[:], signature); err != nil {

        return &CryptoError{

            Type:    VerifyErr,

            Err:     err,

            Content: "signature verification failed",

        }

    }

    return nil

}

func (s \*RSASigner) IsCertValid(cert \*x509.Certificate) bool {

    now := time.Now()

    return now.After(cert.NotBefore) && now.Before(cert.NotAfter)

}

func (s \*RSASigner) CertMarshal(cert \*x509.Certificate) []byte {

    return pem.EncodeToMemory(&pem.Block{

        Type:  "CERTIFICATE",

        Bytes: cert.Raw,

    })

}

func (s \*RSASigner) KeyMarshal(key \*rsa.PrivateKey) []byte {

    return pem.EncodeToMemory(&pem.Block{

        Type:  "RSA PRIVATE KEY",

        Bytes: x509.MarshalPKCS1PrivateKey(key),

    })

}

func (s \*RSASigner) CertUnmarshal(data []byte) (\*x509.Certificate, error) {

    if len(data) == 0 {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     errors.New("empty data"),

            Content: "certificate data cannot be empty",

        }

    }

    block, \_ := pem.Decode(data)

    if block == nil || block.Type != "CERTIFICATE" {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     errors.New("invalid PEM block"),

            Content: "failed to decode PEM block containing certificate",

        }

    }

    cert, err := x509.ParseCertificate(block.Bytes)

    if err != nil {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     err,

            Content: "failed to parse certificate",

        }

    }

    return cert, nil

}

func (s \*RSASigner) KeyUnmarshal(data []byte) (\*rsa.PrivateKey, error) {

    if len(data) == 0 {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     errors.New("empty data"),

            Content: "key data cannot be empty",

        }

    }

    block, \_ := pem.Decode(data)

    if block == nil || block.Type != "RSA PRIVATE KEY" {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     errors.New("invalid PEM block"),

            Content: "failed to decode PEM block containing private key",

        }

    }

    key, err := x509.ParsePKCS1PrivateKey(block.Bytes)

    if err != nil {

        return nil, &CryptoError{

            Type:    DataErr,

            Err:     err,

            Content: "failed to parse private key",

        }

    }

    return key, nil

}

**Network**

package network

import (

    "archive/zip"

    "bytes"

    "crypto"

    "crypto/rand"

    "crypto/rsa"

    "crypto/sha256"

    "crypto/x509"

    "encoding/base64"

    "io"

    "log"

    "net/http"

    "github.com/gin-gonic/gin"

)

var (

    currentCert \*x509.Certificate

    currentKey  \*rsa.PrivateKey

)

type GenerateResponse struct {

    Status string `json:"status"`

}

type SignRequest struct {

    Data []byte `form:"data"`

    Key  []byte `form:"key"`

}

type SignResponse struct {

    Signature []byte `json:"signature"`

}

type VerifyRequest struct {

    Data      []byte `form:"data"`

    Signature []byte `form:"signature"`

    Cert      []byte `form:"cert"`

}

func (r \*Router) generateHandler(c \*gin.Context) {

    cert, key, err := r.signer.GenCertAndKey()

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка генерации ключей",

            "details": err.Error(),

        })

        return

    }

    certPEM := r.signer.CertMarshal(cert)

    keyPEM := r.signer.KeyMarshal(key)

    buf := new(bytes.Buffer)

    zipWriter := zip.NewWriter(buf)

    addFileToZip(zipWriter, "certificate.pem", certPEM)

    addFileToZip(zipWriter, "private\_key.pem", keyPEM)

    if err := zipWriter.Close(); err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка создания архива",

            "details": err.Error(),

        })

        return

    }

    c.Header("Content-Type", "application/zip")

    c.Header("Content-Disposition", "attachment; filename=credentials.zip")

    c.Data(http.StatusOK, "application/zip", buf.Bytes())

}

func addFileToZip(zipWriter \*zip.Writer, filename string, data []byte) error {

    writer, err := zipWriter.Create(filename)

    if err != nil {

        return err

    }

    \_, err = writer.Write(data)

    return err

}

func (r \*Router) signHandler(c \*gin.Context) {

    dataFile, err := c.FormFile("data")

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Необходимо загрузить файл данных",

            "details": err.Error(),

        })

        return

    }

    file, err := dataFile.Open()

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка чтения файла данных",

            "details": err.Error(),

        })

        return

    }

    defer file.Close()

    hasher := sha256.New()

    if \_, err := io.Copy(hasher, file); err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка вычисления хеша",

            "details": err.Error(),

        })

        return

    }

    hash := hasher.Sum(nil)

    keyContent := c.PostForm("key")

    key, err := r.signer.KeyUnmarshal([]byte(keyContent))

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Неверный формат приватного ключа",

            "details": err.Error(),

        })

        return

    }

    signature, err := rsa.SignPKCS1v15(rand.Reader, key, crypto.SHA256, hash)

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка создания подписи",

            "details": err.Error(),

        })

        return

    }

    c.JSON(http.StatusOK, gin.H{

        "signature": base64.StdEncoding.EncodeToString(signature),

    })

}

func (r \*Router) verifyHandler(c \*gin.Context) {

    dataFile, err := c.FormFile("data")

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Необходимо загрузить исходный файл",

            "details": err.Error(),

        })

        return

    }

    file, err := dataFile.Open()

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка чтения исходного файла",

            "details": err.Error(),

        })

        return

    }

    defer file.Close()

    hasher := sha256.New()

    if \_, err := io.Copy(hasher, file); err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error":   "Ошибка вычисления хеша",

            "details": err.Error(),

        })

        return

    }

    hash := hasher.Sum(nil)

    signatureContent := c.PostForm("signature")

    if signatureContent == "" {

        c.JSON(http.StatusBadRequest, gin.H{

            "error": "Необходимо предоставить подпись",

        })

        return

    }

    signature, err := base64.StdEncoding.DecodeString(signatureContent)

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Неверный формат подписи (ожидается base64)",

            "details": err.Error(),

        })

        return

    }

    if len(signature) != 256 { // For RSA-2048

        c.JSON(http.StatusBadRequest, gin.H{

            "error":    "Неверный размер подписи",

            "expected": 256,

            "actual":   len(signature),

        })

        return

    }

    certContent := c.PostForm("cert")

    cert, err := r.signer.CertUnmarshal([]byte(certContent))

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Неверный формат сертификата",

            "details": err.Error(),

        })

        return

    }

    pubKey := cert.PublicKey.(\*rsa.PublicKey)

    if err := rsa.VerifyPKCS1v15(pubKey, crypto.SHA256, hash, signature); err != nil {

        log.Printf("Verify failed: hash=%x signature=%x...", hash, signature[:16])

        c.JSON(http.StatusBadRequest, gin.H{

            "error":   "Проверка подписи не пройдена",

            "details": err.Error(),

        })

        return

    }

    c.JSON(http.StatusOK, gin.H{

        "message": "Подпись верна!",

    })

}

**Router**

package network

import (

    "github.com/gin-gonic/gin"

    "github.com/koan6gi/go-digisign/internal/crypto"

)

type Router struct {

    engine \*gin.Engine

    signer crypto.Signer

}

func NewRouter(signer crypto.Signer) \*Router {

    r := &Router{

        engine: gin.Default(),

        signer: signer,

    }

    r.setupRoutes()

    return r

}

func (r \*Router) setupRoutes() {

    r.engine.Static("/static", "./static")

    r.engine.StaticFile("/", "./static/index.html")

    r.engine.StaticFile("/generate", "./static/generate.html")

    r.engine.StaticFile("/sign", "./static/sign.html")

    r.engine.StaticFile("/verify", "./static/verify.html")

    r.engine.GET("/api/generate", r.generateHandler)

    r.engine.POST("/api/sign", r.signHandler)

    r.engine.POST("/api/verify", r.verifyHandler)

}

func (r \*Router) Run(addr string) error {

    return r.engine.Run(addr)

}