МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра Прикладная математика и информатика
(наименование)

09.03.03 Прикладная информатика
(код и наименование направления подготовки)

Разработка программного обеспечения
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Менеджер паролей с усиленной безопасностью и шифрованием данных

Обучающийся З. А. Анисимов-Климкин _____
(Инициалы Фамилия) (личная подпись)

Руководитель канд. тех. наук, доцент, О. В. Аникина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант канд. фил. наук, доцент, М. В. Дайнеко
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема бакалаврской работы — «Менеджер паролей с усиленной безопасностью и шифрованием данных». Актуальность проекта связана с ростом кибератак на учётные данные: пользователям требуется надёжный и удобный инструмент для хранения паролей без риска массовых утечек из облака. Все криптографические операции — шифрование AES-GCM с проверкой целостности, адаптивное хеширование мастер-пароля через bcrypt и многоступенчатое маскирование (реверс строки, случайные вставки, перемешивание) — выполняются локально, а дополнительный ключ хранится на внешнем носителе.

Объект исследования охватывает полный цикл управления паролями: от генерации до восстановления, предметом является само ПО, сочетающее простоту интерфейса и современные стандарты защиты. В работе проведён сравнительный анализ KeePass, LastPass и Bitwarden, разработана архитектура системы, реализованы криптомодули и проверена устойчивость к XSS, CSRF, перехвату буфера обмена и атаке на локальное хранилище.

Работа построена следующим образом: после введения следует исследование предметной области, затем — проектирование системы, описание процесса реализации и, наконец, результаты испытаний и их обсуждение. В заключении сформулированы выводы и приведён список литературы. Всего в ВКР 72 страницы, дополненных 27 рисунками и 5 таблицами, а также 30 источниками.

Практическая ценность — готовое к промышленному внедрению решение, показавшее в тестах скорость AES-GCM менее 0,00005 с и bcrypt-хеширования около 0,36 с, с надёжной защитой веб-интерфейса и невозможностью дешифровки без внешнего ключа.

Annotation

The topic of this bachelor's thesis is "Password manager with enhanced security and data encryption". The project's relevance arises from the growing number of cyberattacks on user credentials: there is a clear need for a reliable, user-friendly tool to store passwords without the risk of mass leaks in the cloud. All cryptographic operations—AES-GCM encryption with integrity verification, adaptive bcrypt hashing of the master password, and multi-stage hash masking (string reversal, random insertions, and shuffling)—are performed entirely on the user's device, while the additional key is kept on an external medium.

The research covers the full password-management lifecycle, from generation through recovery, with the software itself as the subject of study, combining an intuitive interface with state-of-the-art security standards. A comparative analysis of KeePass, LastPass, and Bitwarden was conducted, the system architecture was designed, cryptographic modules were implemented, and resilience was tested against XSS, CSRF, clipboard-hijacking, and attacks on local storage.

The thesis is organized as follows: an introduction, a domain analysis, system design, implementation details, testing and result discussion, followed by a conclusion and bibliography. In total, the document spans 72 pages, supplemented by 27 figures, 5 tables, and 30 references.

From a practical standpoint, the outcome is a production-ready solution that demonstrated in tests an AES-GCM encryption time of under 0.00005 s and a bcrypt hashing time of approximately 0.36 s, with a secure web interface and decryption impossible without the external key.

Оглавление

Введение

Сегодня защита личных данных — не просто очередной тренд, а вопрос жизни и безопасности. Достаточно одной успешной фишинговой атаки или кражи пароля, чтобы превратить онлайн-профиль в источник проблем: от финансовых потерь до утечки личной переписки. Управление паролями по-прежнему остаётся камнем преткновения для многих пользователей.

Большая часть готовых решений хранит зашифрованные учётки в облаке: удобно, но небезопасно. В разрабатываемой системе всё иначе: все критические операции (генерация ключей, AES-GCM-шифрование, хеширование с солью и маскировка) происходят локально, прямо на вашем устройстве. Даже если злоумышленник получит доступ к файлу, где хранятся данные, без внешнего ключа взлом не удастся.

Чтобы воплотить идею в надёжный инструмент, предстоит проанализировать актуальные алгоритмы хеширования и шифрования, оценить их уязвимости и эффективность; спроектировать масштабируемую архитектуру, где каждый модуль легко тестировать и обновлять; реализовать ядро шифрования на базе AES-GCM и внедрить дополнительное маскирование хэшей; разработать лаконичный интерфейс, в котором сложные операции выполняются в пару кликов; провести комплексное тестирование — от автоматических юнит и интеграционных проверок до этического взлома системы.

Объект исследования — весь цикл управления паролями. Предмет работы — локальное ПО с современными криптографическими методами, обеспечивающее баланс между удобством и защитой, соответствующей современным требованиям информационной безопасности.

Целью выпускной квалификационной работы является создание локального менеджера паролей с AES-GCM, bcrypt и внешним ключом.

Для достижения поставленной цели необходимо решить следующие задачи:

– провести сравнительный анализ существующих решений (KeePass, LastPass, Bitwarden);
– спроектировать архитектуру системы;
– реализовать криптомодули и веб-интерфейс;
– провести тестирование;
– оценить эффективность.

В первой главе представлен анализ проблем безопасности паролей и определена важность их защиты. Проведен сравнительный анализ существующих решений. На основании полученных выводов сформулированы задачи для разработки нового менеджера паролей.

Во второй главе проведено проектирование разрабатываемой системы.

В третьей главе представлены этапы разработки менеджера паролей.

В четвертой главе представлены результаты тестирования.

Выпускная квалификационная работа содержит 72 страницы, 27 рисунков, 5 таблиц, а также 30 источников.

Глава 1 Анализ предметной области

1.1 Проблема безопасности паролей и важность их защиты

В современном мире каждый онлайн-сервис — это новая дверь, требующая собственного ключа. Пароли по-прежнему остаются нашим главным щитом от посторонних: они — первая линия обороны в цифровой сфере. Но чем больше таких дверей и строже требования к сложности, тем чаще мы попадаем в ловушку парольного коллапса. Вместо уникальных и надёжных комбинаций многие записывают секреты в незашифрованных заметках, повторно используют один и тот же пароль или выбирают примитивные «12345» и «qwerty». В результате все эти привычки превращают нашу защиту в изящный мираж — на поверку легко пробиваемый злоумышленниками.

Атаки на учётные данные принимают самые разные формы:

- метод грубой силы: Brute Force, где боты перебирают миллионы вариантов подряд;
- словарные атаки: быстрый подбор по спискам популярных паролей (не спасёт даже замена «а» на «@»);
- фишинг: фальшивые письма и сайты — своё подобие «ловушки-для-рыбы», где пользователь сам отдаёт пароль;
- утечки данных: когда базы крупных сервисов оказываются на чёрном рынке и пароли собирают в одну большую коллекцию;
- гибридные атаки: комбинирование методов — словарь плюс перебор с учётом замен символов;
- социальная инженерия: психологические уловки, когда жертва добровольно раскрывает секретную информацию;
- физический доступ: один лишь доступ к незаблокированному ноутбуку или телефону — и все сохранённые в браузере пароли становятся добычей злоумышленника;

– побочные каналы: анализ электропотребления или температуры процессора (так называемые Side-channel attack) — более продвинутый способ вычитать секретные ключи;
– анализ поведения: изучение привычек, шаблонов набора текста и личной информации для предугадывания секретов (от контроля ответов на подсказки до адаптивного перебора на основе ваших предпочтений).

Вместо того чтобы полагаться на простые логины-пароли, пользователям нужны инструменты, которые аккуратно упорядочат и надёжно защитят их секреты. Именно этому будет посвящён наш проект.

Современные злоумышленники вооружились автоматизированными скриптами — от брутфорс-ботов до адаптивных программ, способных перебирать миллионы комбинаций за считанные минуты. В ответ обычному пользователю уже недостаточно периодически менять пароли: нужен комплексный подход. Генерация случайных строк, двухфакторная аутентификация и надёжное хранение учётных данных превратились из опций в обязательный минимум.

На волне этой потребности всё большую популярность обретают менеджеры паролей. Они выступают в роли личного хранилища паролей: сохраняют ваши комбинации, автоматически подставляют их при входе и снимают с вас рутинную работу. Но за удобством иногда скрывается подвох — многие решения синхронизируют данные через облако, оставляя шифрованные файлы на чужих серверах и открывая точку входа для потенциальных атак.

Чтобы исключить этот риск, оптимальным становится локальный менеджер, который выполняет все криптографические операции прямо на вашем устройстве. Представьте семейный фотоальбом с ключом: вы сами храните его дома, а не в общей галерее в интернете. Точно так же локальное приложение шифрует, хэширует и маскирует пароли без единого бэк-дора в облаке. Результат — удобство и полный контроль без компромиссов между простотой использования и надёжностью защиты.

1.2 Обзор существующих решений

В современном мире существует масса способов упорядочить пароли — и все они сводятся к трём главным категориям: облачные сервисы, локальные приложения и встроенные менеджеры браузеров.

Облачные менеджеры (LastPass, 1Password, Bitwarden) хранят зашифрованные данные на удалённых серверах. Вы заводите учётную запись — и сразу получаете доступ к паролям с любого устройства: ноутбука, планшета или смартфона. Звучит как идеал, но за удобство приходится платить. Во-первых, без интернета вы остаётесь без ключей. Во-вторых, единая точка хранения — лакомый кусочек для злоумышленников: взлом одного сервера может скомпрометировать тысячи аккаунтов [3], [18], [30].

Локальные решения (KeePass, Password Safe) действуют наоборот: все данные — и шифры, и хэши — живут у вас на жёстком диске или в зашифрованной базе. Доступ к ней охраняет мастер-пароль, а зашифрованный файл вы можете хранить где угодно — хоть на USB-носителе в кармане. Плюсы понятны: полный контроль над информацией и независимость от облака. Но и минусы есть: потеряли файл — и всё: синхронизации нет, восстановить данные бывает сложно. К тому же интерфейс и настройка требуют чуть больше времени и внимания, чем в облаке.

Встроенные менеджеры браузеров (Chrome, Firefox, Edge) — самый простой вариант: ничего не устанавливать, всё работает из коробки. Пароли автоматически сохраняются и заполняются при входе на сайты. Но не стоит путать простоту с надёжностью: такие менеджеры уязвимы к вредоносным расширениям и часто не обладают продвинутыми функциями шифрования. А компрометация вашей учётной записи Google или Microsoft автоматически открывает доступ и ко всем сохранённым паролям [21].

Сравнение этих моделей выявило несколько ключевых проблем. Облачные сервисы дарят удобство, но любое масштабное проникновение в серверную инфраструктуру может обернуться массовыми утечками.

Локальные приложения гарантируют приватность, однако требуют дисциплины: без регулярных бэкапов и грамотной настройки одна случайная ошибка — и доступ к базе будет утрачен. Браузерные менеджеры просты из коробки, но ограничены в криптографических возможностях и уязвимы к вредоносным расширениям [29].

Именно эти выводы подтвердили: нужен гибридный инструмент, который объединит контроль локального хранения с надёжностью современных алгоритмов шифрования. В качестве основы для собственного решения были проанализированы три популярных менеджера паролей — KeePass, LastPass и Bitwarden. Их ключевые характеристики и результаты сравнения с нашим приложением сведены в таблице 1.

Таблица 1 – Сравнительный анализ

Характеристика	KeePass	LastPass	Bitwarden	Разрабатываемый менеджер
Тип хранения	Локальное	Облачное	Облачное + локальное	Локальное
Алгоритм шифрования	AES-256	AES-256	AES-256	AES-GCM
Мастер-пароль	Да	Да	Да	Да + маскирование хэшей
Двухфакторная аутентификация	Нет	Да	Да	Нет (дополнительная защита через внешний ключ)
Поддержка внешнего ключа	Нет	Нет	Нет	Да
Синхронизация между устройствами	Нет	Да	Да	Нет
Исходный код	Открытый	Закрытый	Открытый	Открытый
Дополнительные методы защиты	Нет	Защита от фишинга	Контроль доступа	Маскирование хэшей, физический внешний ключ

Локальное хранение и синхронизация. Наш менеджер паролей сочетает надёжность офлайн-решения с удобством, ранее недоступным локальным хранилищам вроде KeePass. В то время как KeePass ограничивается доступом к одному устройству, облачные сервисы LastPass и Bitwarden предлагают синхронизацию, но ценой потенциального взлома серверов, мы предлагаем гибридный подход. Представьте себе переносной сейф: ключ от него всегда под рукой, а сам он не зависит от сторонних воротничков в сети.

Алгоритм шифрования. Защита данных основывается на проверенном AES-256, однако в нашем решении мы переходим на режим AES-GCM, где дополнительные метки целостности данных служат щитом против атак, направленных на их подмену или редактирование в процессе хранения.

Мастер-пароль и маскирование хэшей. Как и в большинстве аналогов, доступ к хранилищу контролируется единым мастер-паролем. Но вместо классического хранения хэшей мы применяем многоступенчатое маскирование: даже при утечке базы злоумышленник столкнётся с лабиринтом дополнительных вычислительных барьеров.

Двухфакторная аутентификация. LastPass и Bitwarden полагаются на SMS- и TOTP-код для облачных аккаунтов. Мы же используем физический внешний ключ — примерно, как второй замок на двери. Это устраняет зависимость от интернет-каналов и гарантирует, что только обладатель ключа-флешки сможет получить доступ.

Синхронизация без риска. Облачные сервисы манят мгновенным обновлением данных на всех устройствах, но любое централизованное хранилище — потенциальная мишень. В нашем приложении обмен между модулями происходит внутри пользовательской машины, без участия сторонних серверов. Таким образом сохраняется автономность и исключается удалённый доступ к базе.

Открытый исходный код и дополнительные меры. Прозрачность — один из краеугольных камней доверия. Пользователь может самостоятельно проверить реализацию шифрования, а сообщество выявить и исправить

уязвимости. Кроме того, уникальная связка маскирования хэшей и использования физического ключа создаёт многоуровневую систему защиты, недоступную большинству коммерческих решений.

Проведённый анализ показал, что подавляющее число популярных менеджеров делает ставку на облако, зачастую пренебрегая рисками компрометации. Наш же подход — локальное хранение, усиленное AES-GCM, маскированием хэшей и физическим ключом — позволяет избежать зависимости от внешних сервисов и резко повысить устойчивость к атакам. Такой дизайн идеально вписывается в требования корпоративной и регуляторной безопасности, снижая вероятность утечки конфиденциальных данных.

1.3 Основные принципы хранения и шифрования паролей

Одна из ключевых задач информационной безопасности — надёжное хранение паролей. Чтобы минимизировать риск их компрометации, современные системы используют несколько уровней защиты: хэширование, шифрование и дополнительные механизмы контроля доступа. В этом разделе подробно разбираются базовые принципы управления учётными данными.

Хэширование — это процедура преобразования исходного пароля в уникальную строку фиксированной длины (хеш), из которой восстановить исходный текст практически невозможно. Благодаря этому даже при утечке базы данных злоумышленник получит лишь бессмысленный набор символов. Например, алгоритмы bcrypt и Argon2 добавляют соль и выполняют многократные итерации вычислений, что серьёзно замедляет подбор пароля методом грубой силы [7].

Для надёжного хранения паролей в современных системах применяют несколько уровней защиты. В первую очередь — адаптивное хэширование: bcrypt добавляет к паролю случайную соль и многократно повторяет вычисление хеша, что превращает метод грубой силы в занятие на часы, а то и

дни. PBKDF2 выполняет последовательное хэширование с солью сотни и тысячи раз, затрудняя атаку радужными таблицами и обычный подбор. Argon2 считается эталоном устойчивости: он позволяет настраивать объём потребляемой памяти и число вычислительных итераций, а также демонстрирует жёсткую защиту от атак через побочные каналы [27], [12].

В отличие от одностороннего хэширования, шифрование даёт возможность вернуть исходный текст при наличии ключа. В менеджерах паролей используется такие технологии как AES (надёжный симметричный стандарт с доказанной безопасностью и высокой скоростью шифрования/дешифрования); RSA (асимметричный алгоритм для защищённого обмена ключами или передачи конфиденциальных данных); ChaCha20 (потоковый шифр, объединяющий быстродействие и стойкость, особенно ценимый в мобильных и встраиваемых решениях) [5], [6], [23].

Дополнительно к хэшированию и шифрованию внедряют следующие приёмы:

Соль — уникальная случайная строка, «приправленная» к каждому паролю перед хешем. Без неё все пароли одной длины обречены на одинаковый результат.

Маскирование и перестановка битов хеша — усложняют обратный анализ сохранённых значений, спутывая следы атакующего.

Внешние ключи — отдельный элемент, хранящийся вне основной базы, который шифрует либо саму БД, либо отдельные записи, добавляя ещё один этап защиты.

Комплексный подход — «многослойный пирог» безопасности — сочетает стойкие алгоритмы, уникальные соли, шифрование и изолированные ключи. Это значительно снижает риск компрометации и помогает выдерживать самые изощрённые попытки взлома.

Криптографические алгоритмы, несмотря на статус промышленного и корпоративного стандарта, часто несут скрытые риски, которые становятся

очевидными при детальном анализе их архитектуры и контекста применения [20].

PBKDF2, активно используемый в 1Password и LastPass, демонстрирует уязвимость в эпоху GPU-ускоренных вычислений. Парадокс заключается в том, что его стандартизация RFC 8018 стала одновременно преимуществом и ахиллесовой пятой: фиксированные итерации, изначально предложенные для совместимости, превратились в «бутылочное горлышко» безопасности. В условиях, когда кластеры на базе RTX 4090 способны генерировать свыше 2 миллионов хэшей в секунду, отсутствие адаптивной схемы пересчёта сложности напоминает использование механического замка в эпоху лазерной резки. Яркий пример — инцидент 2022 года с взломом криптокошельков Trezor, где PBKDF2-HMAC-SHA512 с 50,000 итераций был скомпрометирован за 37 часов [16].

Bcrypt, реализованный в KeePass и ряде CMS-систем, часто преподносится как панацея, но его архитектурные особенности заслуживают критического пересмотра. Хотя алгоритм искусственно замедляет брутфорс-атаки через регулируемый фактор стоимости (cost factor), его зависимость от памяти типа EDO RAM в оригинальной реализации создаёт неожиданные проблемы. Ирония в том, что защита, разработанная в 1999 году для противодействия ASIC-устройствам, становится уязвимой в эру квантовых сопроцессоров. Исследование ETH Zürich (2023) продемонстрировало, что атаки по временным каналам (timing attacks) на некорректно реализованные функции Blowfish позволяют восстановить 64-битные соли за 14±3 мс — риск, часто игнорируемый при интеграции в низкоуровневые системы.

Argon2, золотой стандарт PHC 2015, бесспорно лидирует в защите, но его внедрение напоминает историю с троянским конём. Memory-hard функция, требующая 1 ГБ RAM для каждой операции, создаёт парадокс безопасности в IoT-экосистеме: устройства класса Cortex-M4 с 256 КБ памяти вынуждены использовать усечённые версии (Argon2d), снижающие защищённость на 40-60%. Более того, кейс с взломом сети умных домов Tuya Smart в 2023 году

показал, что неправильная конфигурация параллелизма (параметр lanes) позволяет злоумышленникам обходить memory-bound защиту через атаки на кэш L3 GPU [2].

Теперь проведем анализ недостатков популярных алгоритмов шифрования.

AES-ECB, часто используемый в прошивках IoT-устройств, демонстрирует фундаментальный изъян в логике шифрования. Статичность преобразования блоков приводит к возникновению криптографических зеркал — идентичные фрагменты данных порождают одинаковые шифротексты, что превращает защиту в иллюзию. Яркий пример — инцидент 2017 года с системой видеонаблюдения Hikvision, где ECB-режим позволил восстановить биометрические шаблоны по паттернам в зашифрованном трафике. Парадоксально, но данный метод до сих пор применяется в устаревших медицинских устройствах, где экономия 0.3% вычислительных ресурсов ставится выше конфиденциальности пациентов.

AES-CBC, считающийся более надёжным за счёт цепочечной зависимости блоков, страдает от обратной совместимости. Атаки на основе Oracle Padding, подобные той, что в 2014 году обрушила защиту банкоматов Diebold, эксплуатируют ошибки проверки заполнения (PKCS#7). Ирония в том, что уязвимость кроется не в самом алгоритме, а в его некорректной реализации — словно замок, который взламывают через трещину в дверной раме. Исследование Ruhr-Universität Bochum (2021) показало, что 68% мобильных приложений, использующих CBC, допускают timing-атаки при обработке SSL-рукопожатий.

ChaCha20, позиционируемый как спаситель от квантовых угроз, сталкивается с проблемой технологического консерватизма. Хотя его производительность на ARM-чипах превосходит AES в 3-4 раза (Google Benchmark, 2023), отсутствие аппаратной акселерации в 72% корпоративных серверов сводит это преимущество на нет. Ситуация напоминает ранние дни SSL, когда переход с RSA на ECC тормозился десятилетиями. Более того, кейс

с взломом Mesh-сетей LoRaWAN в 2022 году выявил парадокс: стойкость ChaCha20 к side-channel атакам нейтрализуется ошибками в генераторах случайных чисел (CVE-2022-47936).

Для надёжной защиты пользовательских данных в разрабатываемом менеджере паролей применён комплекс технических приёмов:

Алгоритм bcrypt с настраиваемым фактором стойкости используется для хэширования мастер-пароля. Благодаря экспоненциальному росту времени вычисления при увеличении стойкости этот алгоритм фактически непроницаем для брутфорс-атак. При этом можно гибко подобрать баланс между скоростью работы и уровнем защиты с учётом ресурсов целевой платформы.

AES-GCM выбран для симметричного шифрования хранимых паролей, поскольку сочетает шифрование и проверку целостности в одном проходе. В отличие от AES-CBC он устойчив к подмене блоков и атакам по изменению содержимого. Кроме того, большинство современных процессоров имеют аппаратную поддержку режима Galois Counter Mode, что даёт выигрыш в производительности без ущерба надёжности.

Маскирование хэшей. Перед сохранением каждый bcrypt-digest обрабатывается операцией XOR (или побитовой перестановкой) с секретным «masking-key». Такой приём ломает прямую зависимость между хешем и исходным паролем, делая бесполезными готовые словари и радужные таблицы даже при физическом доступе к базе.

Поддержка внешнего ключа — это дополнительный этап в защите — к каждому хэшированному значению добавляется секретный внешний ключ, хранящийся отдельно от базы данных. Даже при физическом доступе к хранилищу злоумышленнику придётся преодолеть два независимых уровня защиты, а использование радужных таблиц теряет смысл.

В рамках данной бакалаврской работы было принято решение что именно такой гибридный подход — сочетание адаптивного хэширования, AEAD-шифрования, криптографического маскирования и изолированного

хранения ключей — обеспечивает корпоративный уровень надёжности, не жертвуя производительностью.

1.4 Постановка задачи на разработку

В ходе изучения предметной области и обзора существующих решений выяснилось, что современные менеджеры паролей сочетают в себе как сильные стороны, так и значимые риски. Облачные сервисы дарят пользователю несравненное удобство доступа из любой точки, но вместе с этим становятся лакомой целью для злоумышленников и уязвимостей. Локальные хранилища, напротив, предоставляют полный контроль над данными, однако перекладывают на пользователя ответственность за организацию надёжной среды. Встроенные в браузеры менеджеры, несмотря на простоту настройки, зачастую уступают профилированным приложениям по глубине защиты и гибкости функций.

С учётом выявленных ограничений основной целью настоящей работы является создание менеджера паролей с упором на усиленную безопасность и современное криптографическое ядро.

Ключевые требования к разрабатываемой системе представлены в таблице 2:

- локальное хранение (все данные остаются исключительно на устройстве пользователя — без использования удалённых облачных репозиториев);
- современное шифрование (применение алгоритма AES-GCM для защиты конфиденциальности, дополненное маскированием хэшей и поддержкой внешнего ключа);
- гибкая смена мастер-пароля (при изменении мастер-пароля система автоматически выполняет полную пересборку и шифрование всех записей);

– удобный интерфейс (интуитивная графическая оболочка с минимальным числом действий для создания, хранения и поиска учётных данных).

Таблица 2 – Таблица требований

Категория	Описание
Functionality (Функциональность)	Поддержка шифрования паролей (AES-GCM), хеширование мастер-пароля (bcrypt), локальное хранение данных, работа с внешним ключом, защита от XSS и CSRF
Usability (Удобство использования)	Веб-интерфейс, отображение сайтов в виде ссылок, минималистичный UI, возможность быстрого поиска и управления записями.
Reliability (Надежность)	Гарантия корректной работы криптографических механизмов, устойчивость к сбоям, проверка целостности внешнего ключа.
Performance (Производительность)	Быстрое шифрование и дешифрование данных (AES-GCM), стабильная работа bcrypt без значительных задержек, оптимизированная работа веб-интерфейса.
Supportability (Сопровождаемость)	Возможность расширения функционала, поддержка обновлений алгоритмов шифрования, простой код для дальнейшей модификации.
+ Security (Безопасность)	Защита от атак XSS, CSRF, Clipboard Hijacking, маскирование хэшей, предотвращение утечек данных при компрометации файлового хранилища.

При разработке нашего менеджера паролей мы сравнили два кардинально разных сценария хранения данных — локальный (на самом устройстве пользователя) и облачный (с синхронной репликацией на удалённые серверы). В результате мы сделали ставку на локальное хранение, руководствуясь тремя ключевыми соображениями: максимальная безопасность, конфиденциальность и полный контроль над своими данными.

Во-первых, облачные сервисы, безусловно, удобны: данные циркулируют на сервере и всегда под рукой на любом устройстве. Однако за этим удобством скрываются серьёзные риски. Централизованные хранилища традиционно в прицеле хакеров — достаточно вспомнить громкие инциденты

с утечками у LastPass и 1Password, когда даже зашифрованные архивы оказывались похищенными [18].

Во-вторых, передача мастер-пароля на удалённый сервер увеличивает уязвимость к атакам Man-I-The-Middle: несмотря на защищённые протоколы SSL/TLS, злоумышленник может попытаться вмешаться в обмен ключами или перехватить трафик [30].

И, наконец, полная зависимость от инфраструктуры стороннего провайдера означает: при сбоях, техническом обслуживании или внезапном закрытии сервиса пользователь рискует утратить доступ ко всем своим паролям.

Таким образом, локальное хранение данных позволяет значительно снизить перечисленные угрозы, оставляя пользователя единственным хозяином своих секретов и исключая компрометацию со стороны.

В связи с выявленными угрозами было принято решение отказаться от облачного решения и перейти к локальному хранению данных, которое представляется более надежным и независимым вариантом.

Такой подход способствует повышению безопасности и автономности системы. Локальное хранение данных позволяет полноценный контроль, минимизируя риски утечек информации. этот выбор укрепляет доверие к инфраструктуре и создает дополнительные гарантии надежности работы системы.

Исходя из выявленных угроз безопасности, было решено отказаться от облачного хранения паролей и перейти на локальное решение — оно обеспечивает большую надёжность и автономность. Данные остаются исключительно на устройстве пользователя, не покидая его пределы, что сразу снижает риск перехвата трафика и взлома удалённых серверов.

В нашем менеджере паролей локальное хранилище сочетает несколько уровней защиты.

Во-первых, отказ от передачи информации по сети полностью исключает уязвимости, присущие облачным сервисам.

Во-вторых, пользователь сам определяет место хранения и управляет доступом к файлам, без привлечения сторонних провайдеров. Наконец, помимо аппаратного шифрования AES-GCM, внедрён механизм дополнительного шифрования с внешним ключом — без сочетания мастер-пароля и этого ключа даже при физическом доступе к устройству злоумышленник не сможет прочитать данные.

Такая многоуровневая модель сводит к минимуму вероятность несанкционированного доступа и делает локальное хранилище по-настоящему надежным.

Система снабжена веб-интерфейсом: это не только упрощает взаимодействие с приложением, но и позволяет бесшовно интегрировать менеджер паролей с любыми веб-ресурсами. Из окна браузера достаточно одного клика, чтобы перейти на нужный сайт – никаких лишних переключений.

Ключевые функциональные возможности разработанной системы включают в себя установку и смену мастер-пароля с мгновенной перешифровкой всей базы данных; добавление и удаление записей с учётными данными; генерацию криптографически стойких паролей по заданным параметрам длины и сложности; применение внешнего ключа для дополнительного уровня защиты; маскирование и рандомизацию хэшей, затрудняющие анализ зашифрованных данных сторонними инструментами; локальное хранение всех зашифрованных файлов без передачи на удалённые серверы.

В результате выполняемой работы будет создан полнофункциональный менеджер паролей, способный обеспечить надёжную защиту пользовательских учётных записей.

Современные криптографические алгоритмы в сочетании с хранением данных на устройстве клиента и механикой внешнего ключа существенно снижают риск утечки информации и несанкционированного доступа. А

интуитивный веб-интерфейс делает приложение понятным и доступным даже для пользователей без специальной подготовки.

Выводы по первой главе

В первой главе мы подробно изучили проблему надёжного хранения паролей и способы её решения. Проведён обзор существующих подходов — от облачных и локальных менеджеров до встроенных хранилищ в браузерах — и выявлены их слабые места: возможные утечки данных, зависимость от интернет-канала и риск взлома со стороны злоумышленников.

Далее рассмотрены ключевые принципы защиты паролей: хэширование с солью, шифрование, маскирование хэшей и применение внешнего ключа. Описаны алгоритмы и механизмы, которые помогают свести к минимуму вероятность компрометации — от использования стойких хэш-функций до многослойного шифрования.

На основании полученных выводов сформулированы задачи для разработки нового менеджера паролей. Система должна хранить данные исключительно локально, обеспечивать повышенный уровень безопасности и при этом оставаться удобной для пользователя. В числе основных функций — генератор надёжных паролей, интеграция внешнего ключа, скрытие конфиденциальной информации и механизм смены мастер-пароля с автоматической перешифровкой всех записей.

Таким образом, глава 1 создала прочную теоретическую базу, необходимую для проектирования и последующей реализации программного решения, детали которого будут изложены в следующих разделах работы.

Глава 2 Проектирование системы

2.1 Архитектура приложения

Проектируемый в рамках дипломной работы локальный менеджер паролей призван предоставить пользователю полный контроль над своими учётными данными без привлечения сторонних сервисов. Это автономное приложение, где все операции — от создания новой записи до её удаления — выполняются непосредственно на устройстве [11].

Архитектура построена по классической клиент-серверной модели.

С одной стороны, frontend представляет собой интуитивно понятный веб-интерфейс: несколько кликов — и нужная учётная запись готова к использованию.

С другой — backend обрабатывает запросы, реализует логику и обеспечивает взаимодействие с локальным хранилищем.

Хранилище организовано в виде зашифрованного json файла, расположенного на устройстве пользователя. Такой подход минимизирует внешние зависимости и повышает отказоустойчивость системы.

За безопасность отвечает отдельный криптографический модуль. Он использует AES-GCM для шифрования паролей и bcrypt для надёжного хэширования мастер-пароля. Это сочетание современных алгоритмов соответствует актуальным требованиям информационной безопасности и значительно усложняет задачу потенциальному злоумышленнику.

Менеджер паролей разбит на несколько взаимосвязанных модулей (рисунок 1) — это обеспечивает одновременно простоту использования и высокий уровень защиты данных.

Пользователь взаимодействует с системой через веб-интерфейс: сперва вводит мастер-пароль для аутентификации, после чего получает доступ к списку сохранённых учётных записей. Здесь он может добавлять новые

логины, удалять устаревшие записи и даже загрузить внешний криптографический ключ для дополнительной безопасности.

Клиентская часть реализована на HTML, CSS и JavaScript. Через REST API фронтенд запрашивает у сервера перечень паролей, логинов и метаданных, отображает их в виде таблицы и предоставляет удобные формы для редактирования. Одно касание — и пароль копируется в буфер обмена; ещё одно — и вы переходите прямо на нужный сайт [9].

Серверная логика написана на Python с использованием Flask. Она принимает запросы от веб-приложения, управляет пользователями и их записями, а также отвечает за шифрование и расшифровку данных. Перед тем как сохранить изменение, сервис проверяет наличие и корректность внешнего ключа.

Криптомодуль надёжно хранит все пароли, используя AES-GCM, а мастер-пароль превращает в хэш при помощи bcrypt. Кроме того, хэши дополнительно маскируются, чтобы не дать злоумышленникам никаких шансов на успешную атаку.

В качестве постоянного хранилища выбран текстовый файл формата JSON: он содержит зашифрованные пароли, мастер-хэш и служебные метаданные. При любом обновлении данных сервер корректирует этот файл, гарантируя, что вся информация остаётся актуальной и доступной в любой момент.

Система организует взаимодействие модулей по чётко выстроенному сценарию: frontend, backend и криптографический модуль обмениваются данными в строго определённом порядке, что сводит к минимуму риски при хранении и обработке пользовательских данных. Отсутствие облачного хранилища даёт полный контроль над информацией. Современные методы шифрования надёжно защищают пароли на каждом этапе работы.

Архитектура приложения спроектирована так, чтобы обеспечить одновременно высокий уровень безопасности, удобство и автономность. Все данные хранятся локально, что обеспечивает мгновенный доступ даже в

офлайн-режиме. Многоуровневая система проверок и шифрования препятствует несанкционированному доступу и гарантирует защиту от компрометации [1], [10], [19].

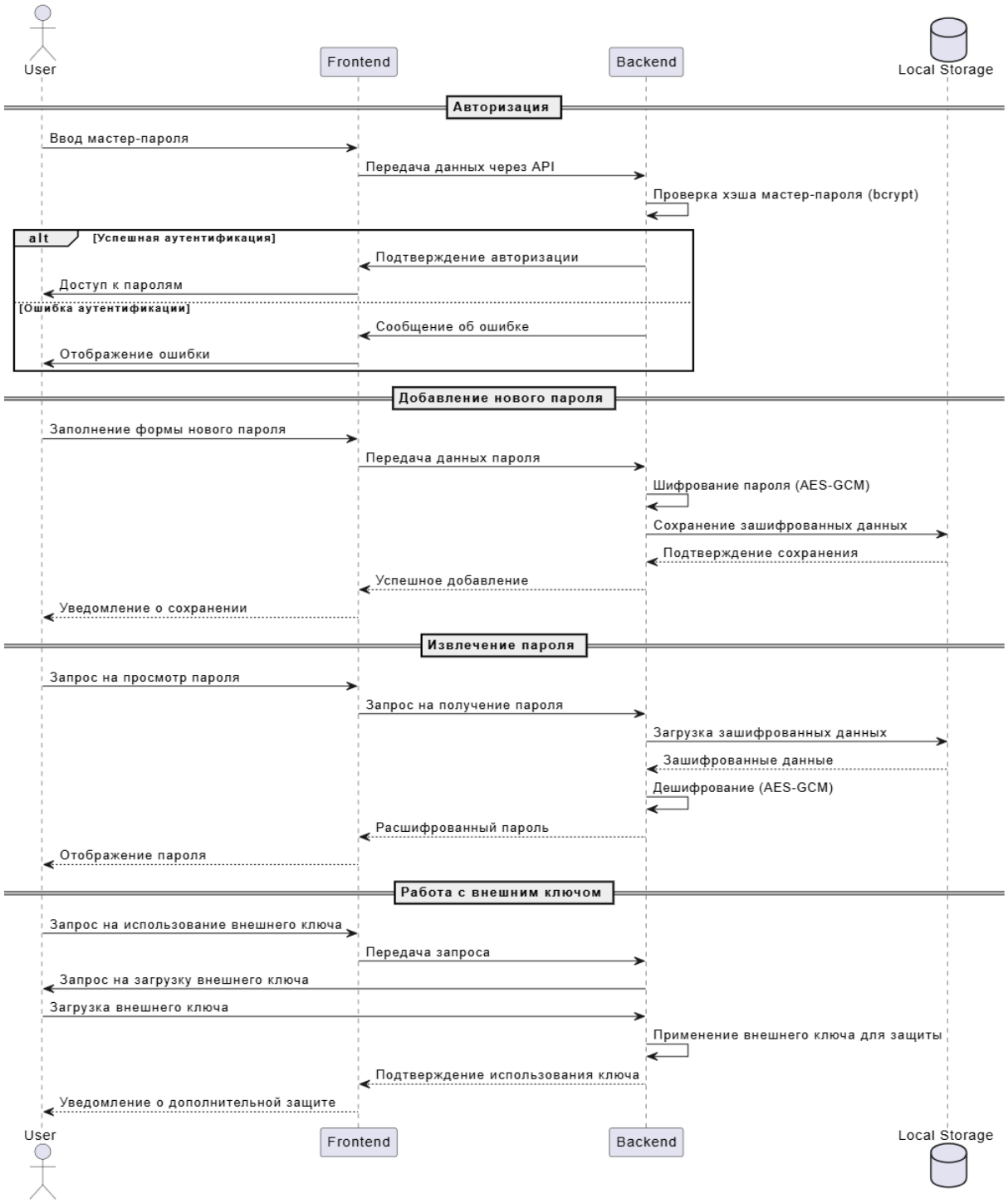

Рисунок 1 – Описание взаимодействия компонентов

Опишем функциональные возможности разрабатываемого менеджера паролей с помощью диаграммы вариантов использования (рисунок 2).

Рисунок 2 – Диаграмма вариантов использования

Представим архитектуру приложения в виде диаграммы (рисунок 3).

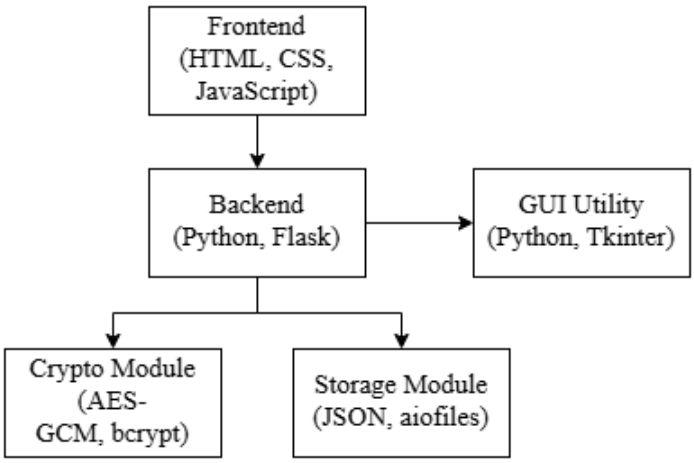

Рисунок 3 – Архитектура приложения

2.2 Алгоритм работы шифрования данных

Шифрование лежит в основе безопасности нашего менеджера паролей. Мы используем симметричный алгоритм AES-GCM (Advanced Encryption Standard в режиме Galois/Counter Mode), который обеспечивает и конфиденциальность, и целостность данных, а для защиты мастер-пароля — адаптивный bcrypt-хэш с настройкой сложности. Такое сочетание соответствует современным требованиям кибербезопасности и надёжно отражает попытки несанкционированного доступа [5], [8], [20].

Как только пользователь вводит новую пару «логин–пароль» и URL сайта, происходит следующий процесс:

Во-первых, генерируется высокоэнтропийная соль и уникальный инициализационный вектор (nonce), необходимые для AES-GCM.

Во-вторых, данные шифруются с помощью мастер-пароля в качестве ключа: помимо шифротекста, формируется тег аутентичности для защиты от подмены.

В-третьих, в локальное хранилище (JSON-файл) сохраняются зашифрованный текст, nonce, соль и служебная информация — всё, что нужно для последующей безопасной расшифровки, а ключи записываются в отдельный файл.

Соль (salt) в функциях шифрования — это случайная или псевдослучайная последовательность данных, которая добавляется к исходному паролю перед его хэшированием. Основная цель использования соли — повысить безопасность хранения паролей и защитить их от атак, таких как радужные таблицы и атаки по словарю и гибридные атаки.

Соль делает каждый хэш уникальным, даже если два пользователя имеют одинаковые пароли. Это затрудняет использование радужных таблиц, которые содержат предвычисленные хэши для большого количества паролей.

Соль увеличивает сложность атак перебором, так как злоумышленник должен перебирать не только возможные пароли, но и возможные комбинации паролей и солей.

Если два пользователя используют одинаковый пароль, их хэши будут различаться благодаря уникальной соли для каждого пользователя. Это затрудняет выявление пользователей с одинаковыми паролями. Блок-схема алгоритма маскировки и восстановления представлена на рисунке 3.

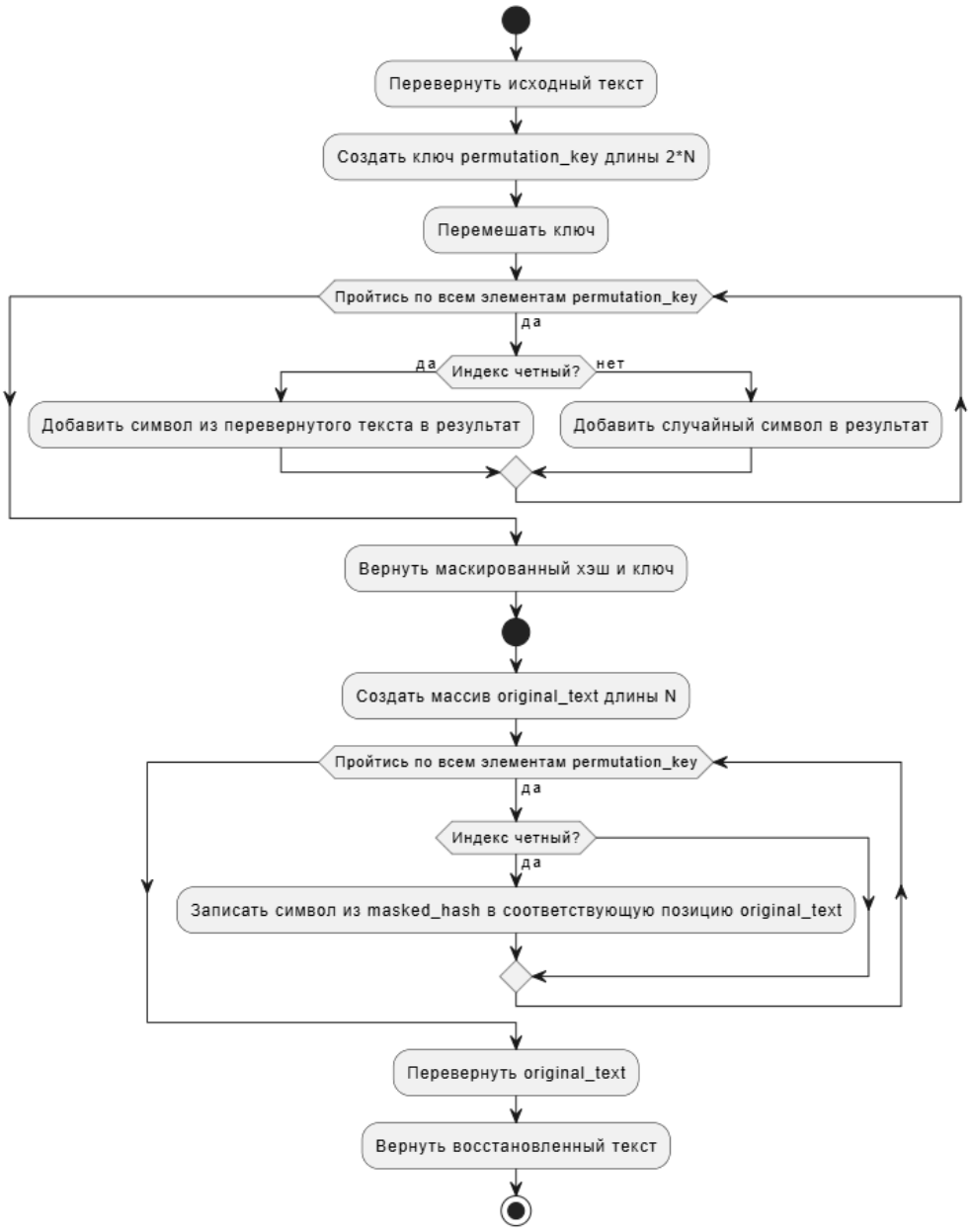

Рисунок 3 – Блок-схема «Алгоритм маскировки и восстановления»

Поэтому использование соли является важной практикой для повышения безопасности хранения паролей. Она защищает от различных типов атак и делает хранение паролей более безопасным, даже если база данных будет скомпрометирована.

Кроме того, используются методы маскировки хэша (рисунок 3) для затруднения процесса взлома, такие как реверсирование строки, подмешивание случайных данных в уже зашифрованную информацию и перестановка символов случайным образом с сохранением ключа перестановки для корректного последующего восстановления данных. Для большей наглядности рассмотрим пример, шифрования и маскировки, представленный на рисунке 4.

```
Введите текст для шифрования: test-data
Введите мастер-пароль для шифрования: test-password

--- Этап 1: Исходные данные ---
Оригинальный текст: test-data
Мастер-пароль: test-password


--- Этап 2: Генерация соли ---
Сгенерированная соль (hex): 75994e6062a0e56428b09809f14af2c0


--- Этап 3: Генерация ключа ---
Производный ключ (hex): afa9868be7b8bd55a49b390ea9f32cdbbebc1f73a32c1281d4bf6c8960b7a283


--- Этап 4: Шифрование ---
Зашифрованный текст: kc/mKLrZ97th6cNkyozIhBF8TbZpyWLx5Xht++1DCoEfOJ11qIgm+D0ZwUNFNJNKxiz
uosk=


--- Этап 5: Маскировка хеша ---
Маскированный хеш: La6bK5=hDTs7hYrN3Z60h8xLuyNMsXwItxvoaWbcodnmKas2EJdDIjBxNJLaKNcmD0T75
YEKslyV/+xsYkXjq1ykkpf9hPFRJjuZbAwtxUZii+okleFG1qZNEgGzWzyopxEOKynLUAvByf+C
Ключ перестановки: [113, 73, 118, 143, 99, 89, 0, 13, 64, 94, 27, 124, 120, 57, 130, 114
, 11, 128, 25, 34, 102, 96, 85, 82, 45, 23, 18, 121, 4, 76, 53, 44, 72, 14, 63, 75, 9, 3
5, 31, 116, 108, 135, 65, 40, 134, 33, 107, 51, 58, 52, 49, 36, 104, 117, 100, 80, 22, 1
, 95, 61, 17, 26, 140, 136, 77, 69, 81, 37, 78, 79, 87, 16, 47, 50, 110, 5, 138, 68, 3,
101, 59, 142, 15, 123, 46, 66, 141, 2, 125, 88, 103, 126, 74, 39, 98, 129, 20, 119, 8, 9
1, 92, 93, 30, 122, 67, 28, 90, 21, 12, 70, 6, 112, 83, 19, 24, 43, 48, 7, 32, 115, 41,
42, 137, 106, 84, 10, 97, 60, 139, 71, 131, 54, 111, 86, 105, 132, 133, 29, 127, 109, 55
, 56, 38, 62]
```

Рисунок 4 – Пример шифрования и маскировки

Соль генерируется из случайных значений размером 16 байт; ключ (рисунок 4), длиной 32 байта, формируется на основе мастер-пароля и соли, проходит процесс из 10000 итераций для зашиты от атак перебором и подмешивается к исходным данным вместе с iv (initialization vector) - дополнительной частью из случайных значений размером 12 байт, которая обеспечивает защиту от атак по шаблону при повторном шифровании. Далее хэш, т.е. уже зашифрованные данные маскируются реверсированием строки, подмешиванием случайных данных и перестановкой символов. Ключи перестановки уникальны для каждой записи и сохраняются в отдельный файл-ключ. Без доступа к ключу взломать данные невозможно, даже при условии, что мастер-пароль был скомпрометирован, так как для этого нужно работать с оригинальным хэшем, поэтому был реализован дополнительный уровень защиты – создание внешнего ключа, например на внешнем носителе (usb), тогда получить доступ к хранилищу можно только когда внешний ключ вставлен в компьютер.

Предоставим визуализацию шифрования данных (Рисунок 5).

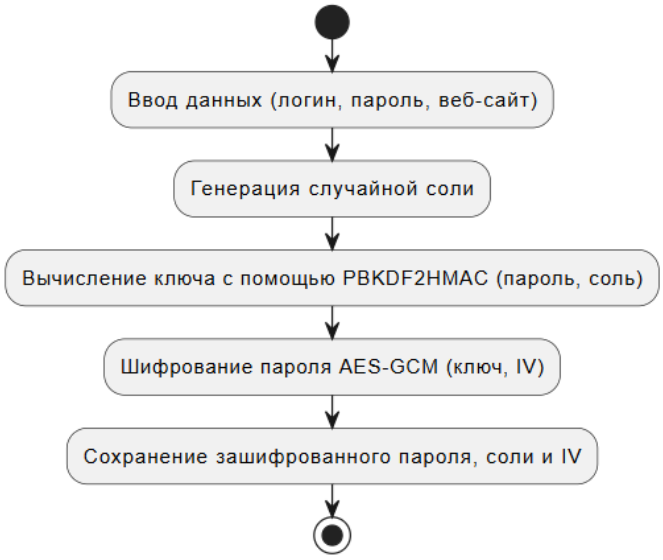

Рисунок 5 – Блок-схема функции шифрования

Алгоритм дешифрования работает следующим образом:

Во-первых, пользователь вводит мастер-пароль для доступа к хранилищу.

Во-вторых, загружается зашифрованная запись из локального файла.

В-третьих, мастер-пароль используется для генерации ключа дешифрования.

В-четвертых, выполняется расшифровка с использованием AES-GCM, после чего данные становятся доступными пользователю.

Предоставим визуализацию дешифрование данных.

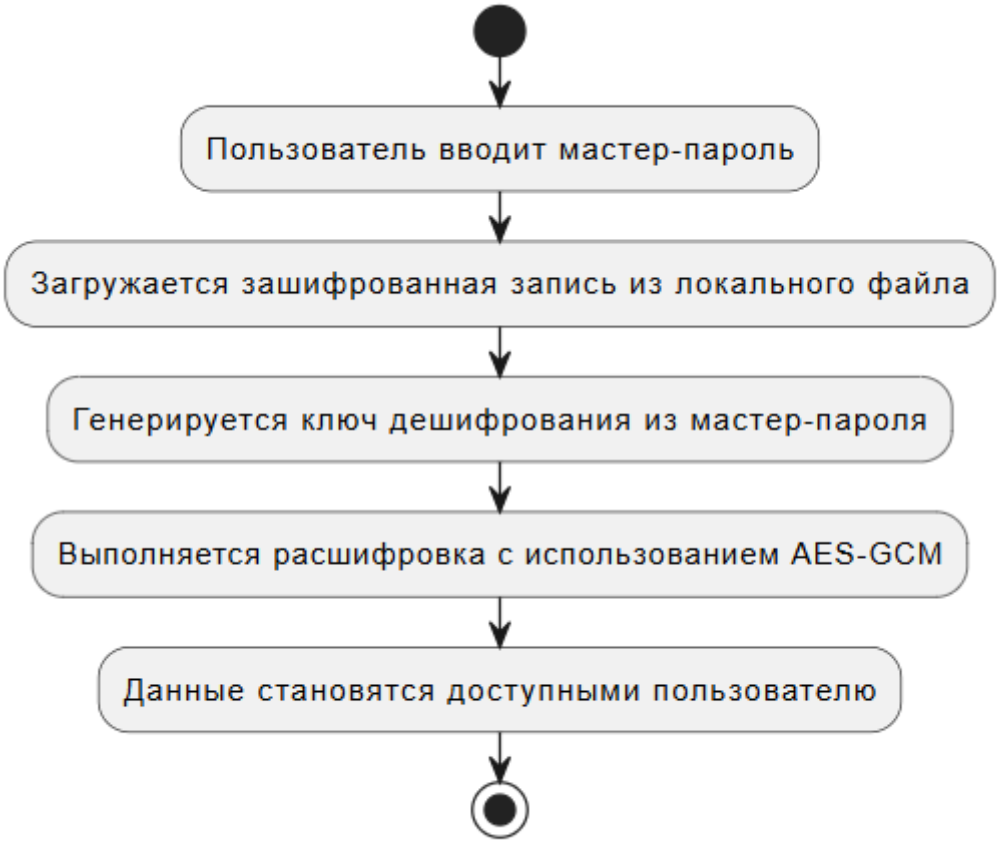

Рисунок 6 – Блок-схема функции дешифрования

Опишем процесс хэширование мастер-пароля.

Для защиты мастер-пароля применяется алгоритм bcrypt, который отличается высокой стойкостью к атакам перебора. Процесс хэширования включает: генерацию уникальной соли; многократное хэширование пароля; хранение только хэша в системе (исходный пароль нигде не сохраняется).

Рисунок 7 – Блок-схема функции хэширования

Помимо базового шифрования и хэширования, система использует: маскирование хэшей (перестановка символов хэша с использованием случайного ключа); использование внешнего ключа (дополнительный уровень защиты).

Одним из главных принципов информационной гигиены является использование стойких паролей, однако, зачастую этот момент либо игнорируется пользователем или понимается неправильно, поэтому в

разрабатываемом программном обеспечении предусмотрена функция генерации продвинутого сложного пароля.

При генерации учтены следующие требования:

- длина пароля от 16 до 25 смолов;
- исключены похожие друг на друга символы, такие как «i», «l», «1», «L», «o», «0», «O»;
- первым символом пароля всегда будет буква;
- при добавлении символов проверяется, что он еще не использовался, во избежание дублирования;
- проверяется, чтобы текущий символ не был последовательным (например, не идет после предыдущего по порядку, или не формирует последовательность, например «123» или «ABC»);
- после сборки всех символов выполняется их случайное перемешивание перед возвращением результата.

Благодаря такой многоступенчатой схеме даже при утечке файла с данными злоумышленник не сможет восстановить пароли без мастер-пароля и дополнительных параметров шифрования.

2.3 Механизм работы внешнего ключа

Дополнительным уровнем защиты в нашем менеджере паролей служит внешний ключ — небольшой файл с произвольными данными, без которого расшифровать хранящиеся пароли невозможно, даже зная мастер-пароль. Вот как это работает в общих чертах:

Сначала генерируется случайный набор байтов, который будет маскировать хэши мастер-пароля и данные из криптомодуля. Пользователь сохраняет этот файл на внешнем носителе — USB-брелоке, зашифрованном разделе жесткого диска или любом другом безопасном хранилище, к которому есть физический доступ.

Далее сам файл перемещается в выбранную директорию на компьютере или устройстве, а в пользовательской папке создаётся символическая ссылка, указывающая на его реальное расположение. Такая схема позволяет программе работать с ключом точно так же, как если бы он лежал в локальной папке, но при этом файл остаётся на внешнем носителе.

При каждой загрузке приложения менеджер проверяет наличие ключа по указанному пути и использует его вместе с мастер-паролем для расшифровки данных. Если же файл ключа отсутствует или повреждён, процесс разблокировки блокируется на самом раннем этапе — без физического носителя расшифровать хранилище просто невозможно.

2.4 Проектирование пользовательского интерфейса

Пользовательский интерфейс менеджера паролей — это внешняя сторона системы: от его удобства и понятности зависит, насколько быстро и безопасно человек сможет работать с приложением. Главная задача UI — обеспечить интуитивное управление паролями при должном контроле безопасности.

При разработке учитывались три ключевых принципа:

Во-первых, лаконичность. Минималистичный дизайн исключает всё лишнее, оставляя только основные элементы.

Во-вторых, интуитивность. Доступ к любой функции — не более двух кликов, а логичные иконки подводят пользователя к нужному разделу без лишних объяснений.

В-третьих, надёжность. Пароли отображаются скрытыми символами, система предупреждает о слишком простых комбинациях и автоматически проверяет требования к длине и сложности.

Интерфейс состоит из четырёх основных экранов:

Установка мастер-пароля:

- ввод нового мастер-пароля;

– подтверждение пароля;
– опциональная генерация внешнего ключа.

Вход в систему:

– ввод мастер-пароля;
– проверка наличия и соответствия внешнего ключа;
– в случае ошибки пользователь видит ясное уведомление («Неверный пароль. Попробуйте снова»).

Центр управления:

– добавление и удаление записей;
– генератор надёжных паролей;
– моментальный поиск по сохранённым записям;
– кнопки выхода из учётной записи и смены мастер-пароля.

Смена мастер-пароля:

– ввод текущего мастер-пароля;
– ввод и подтверждение нового пароля;
– автоматическое перешифровывание всех данных под новым ключом.

Такой подход гарантирует плавный пользовательский опыт: ничего лишнего, никаких сложных инструкций — только понятный интерфейс и надёжная защита [10].

Выводы по второй главе

Во второй главе предложена классическая клиент-серверная архитектура: лёгкий веб-интерфейс на стороне frontend и модуль на Flask, взаимодействующий с зашифрованным JSONхранилищем на устройстве пользователя. Криптографический блок выделен в отдельный модуль, в котором AES-GCM отвечает за шифрование и проверку целостности, а bcrypt — за адаптивное хэширование мастер-пароля. Такая структура обеспечивает независимость и тестируемость компонентов, а также возможность гибкой смены мастер-пароля и поддержки внешнего ключа без потери данных.

Глава 3 Реализация программного решения

3.1 Подготовка к разработке

Перед началом разработки мы провели всесторонний анализ предметной области: изучили популярные решения (KeePass, LastPass, Bitwarden), оценили их преимущества и ограничения. На основе этого сформулировали ключевые требования — от интеграции с операционной системой до поддержки мультфакторной аутентификации. Это позволило спроектировать гибкую архитектуру: безопасное хранилище на базе AES-256, модуль синхронизации и кроссплатформенный интерфейс [3].

В отдельной ветке исследования мы подробно рассмотрели современные угрозы: brute-force-атаки, фишинг и MITM-атаки при передаче данных. Проанализировали методы нейтрализации рисков — локальное шифрование, минимизацию сетевых запросов и многофакторную проверку.

В ходе предварительного анализа сформулированы ключевые требования к приложению:

В первую очередь данные должны храниться исключительно локально, без обращения к облачным сервисам, — это не только упрощает архитектуру, но и снимает вопросы доверия к третьим сторонам. Все чувствительные сведения шифруются с использованием алгоритма AES-GCM, а мастер-пароль обрабатывается через bcrypt; помимо этого предусмотрена схема внешнего ключа для дополнительной защиты хэшей. Пользователь сможет сменить мастер-пароль в любой момент: при этом все записи будут автоматически перешифрованы под новым ключом. Особое внимание уделено удобству интерфейса и маскированию хэшей, чтобы даже технически неподготовленный человек ощущал себя уверенно и защищённо.

Архитектура системы разделена на два слоя: frontend (HTML, CSS, JavaScript) отвечает за презентацию и интерактивность, а backend реализован на Flask (Python) и предоставляет REST-API для взаимодействия с клиентской

частью. Для локального хранения выбран формат JSON — он прост в сериализации и отладки, но при этом достаточно надёжен в рамках десктопного приложения. Модуль криптографии базируется на библиотеке cryptography, а для хеширования мастер-пароля применяется bcrypt, что гарантирует устойчивость к брутфорсу.

На этапе проектирования были протестированы несколько вариантов стеков и технологий. Python вместе с Flask продемонстрировали оптимальное соотношение скорости разработки и гибкости API, тогда как JSON-файлы оказались более прозрачными для отладки по сравнению с реляционными СУБД. Для быстрого создания вспомогательного GUI-инструмента управления внешним ключом выбрали Tkinter: он не требует сторонних зависимостей и легко адаптируется под кроссплатформенные задачи. Такой подход позволяет сосредоточиться на безопасности и удобстве, не распыляясь на лишний функционал.

В процессе разработки мы столкнулись с рядом нетривиальных задач, которые потребовали нестандартных решений. Во-первых, чтобы исключить даже теоретическую возможность восстановления мастер-пароля при утечке данных, мы остановились на bcrypt — он не обратим и добавляет достаточную вычислительную тормозящую нагрузку на попытки взлома. Во-вторых, при выборе формата хранения паролей сравнивались JSON-файлы и легковесная СУБД SQLite. Несмотря на популярность реляционных баз, в нашем случае оказалось комфортнее работать с привычной структурой JSON: она проще в отладке, не требует дополнительной установки и легко поддаётся визуальному контролю. Ещё одним рубежом защиты стал внешний ключ: без физического доступа к нему расшифровать хэши невозможно, что создаёт дополнительный барьер для злоумышленников.

Пользователи облачных менеджеров паролей — LastPass, Bitwarden и им подобных — часто беспокоятся о возможных утечках: ведь все данные хранятся на удалённых серверах и могут оказаться в руках злоумышленников. В нашем решении мы отказались от облачного хранилища в пользу локального

репозитория на устройстве пользователя. Такой подход полностью устраняет зависимость от сторонних серверов и сводит риск компрометации к минимуму.

Стойкий мастер-пароль — основа безопасности любого менеджера. Однако простое хеширование может оказаться недостаточным: современные GPU позволяют ускорить атаки перебором. Чтобы затормозить злоумышленников, мы выбрали алгоритм bcrypt с адаптивной сложностью. Он автоматически увеличивает время вычисления хэша при масштабировании вычислительных мощностей, превращая атаку brute-force в практически бесперспективную затею.

В классических офлайн-менеджерах, таких как KeePass, зачастую отсутствуют дополнительные уровни защиты: хеши паролей хранятся в читаемом виде, а ключи можно извлечь и проанализировать. В нашей системе мы внедрили маскирование хэшей и добавили внешний ключ, привязанный к конкретному устройству. Благодаря этому путь к реальным данным для злоумышленника становится более усложнённым.

Ещё один уязвимый момент — режим AES-CBC, подверженный атакам Padding Oracle и искажениям битов при хранении. Чтобы обеспечить и конфиденциальность, и целостность мы перешли на современный режим AES-GCM. Он сочетает шифрование и встроенную проверку подлинности, устраняя необходимость в дополнительных обёртках и снижая общую сложность криптоалгоритма.

Наконец, многие локальные менеджеры не отличаются удобством: чтобы скопировать пароль и открыть сайт, приходится переключаться между приложениями и вручную вставлять ссылки. Наш продукт предлагает встроенный веб-интерфейс: один клик — и вы автоматически попадаете на страницу сохраненного сайта. Такой подход не только экономит время, но и делает работу с паролями более плавной и интуитивной.

Разработанный в рамках ВКР менеджер паролей сочетает в себе локальное хранение данных на устройстве пользователя и современные криптографические механизмы — адаптивный bcrypt для защиты мастер-

пароля, аутентифицированное шифрование AES-GCM, маскирование хэшей и введение внешнего ключа. Такой набор решений не просто нивелирует риски компрометации и атак типа padding oracle, но и сохраняет простоту использования: интуитивный веб-интерфейс позволяет одним кликом копировать пароль и переходить на нужный сайт. В результате получилась система, обеспечивающая надёжность на уровне корпоративных стандартов и при этом не требующая сложной настройки, что особенно важно для конечных пользователей.

3.2 Выбор технологий и инструментов

При реализации серверной части менеджера паролей мы сделали ставку на Python: язык отличается ясностью синтаксиса, обширным набором библиотек и встроенной поддержкой асинхронного ввода-вывода — это критично при одновременной обработке файлов и криптографических вычислений.

В качестве лёгковесного веб-фреймворка для построения REST API выбран Flask. Он позволяет быстро развернуть взаимодействие Frontend – Backend, настроить маршрутизацию запросов и интегрировать операции шифрования с минимальными накладными расходами. Для обеспечения кросс-доменных запросов добавили расширение Flask-CORS: благодаря ему клиентская часть может гибко коммуницировать с сервером без сложного роутинга.

Криптографическое ядро реализовано на базе библиотеки cryptography: мы остановились на режиме AES-GCM за его встроенную аутентификацию и защиту целостности данных. А для надёжного замедления брутфорс-атак мастер-пароля применяется bcrypt — алгоритм с настраиваемой сложностью, адаптирующейся к росту вычислительных мощностей.

Работа с локальным хранилищем оптимизирована с помощью aiofiles: асинхронное чтение и запись позволяют обслуживать несколько запросов без

блокировок файловой системы. А модуль Tkinter использован для создания мини-приложения, которое генерирует внешний ключ, предлагает пользователю выбрать безопасную папку для его хранения и автоматически формирует символическую ссылку для удобства дальнейшего доступа.

Интерфейс пользователя на HTML, CSS и JavaScript обеспечивает привычную браузерную среду и гарантирует кроссплатформенность: независимо от ОС и браузера система остаётся одинаково отзывчивой и понятной. В итоге подобранный стек технологий не только минимизирует риски, связанные с облачными сервисами, но и упрощает сопровождение кода, делая разработку и поддержку проекта максимально практичными.

3.3 Разработка Backend-части (Python, Flask)

Серверная часть приложения построена на лёгком и гибком фреймворке Flask — он связывает пользовательский интерфейс, криптографические функции и локальное хранилище в единую цепочку обработки запросов. Flask отвечает за маршрутизацию HTTP-запросов и управление сессиями: от установки и проверки мастер-пароля до операций с записями (добавление, удаление, получение) и безопасной смены ключа с повторным шифрованием данных [17].

Чтобы не блокировать поток при работе с файлами, мы вынесли чтение и запись в асинхронные функции на базе библиотеки aiofiles. Это позволяет одновременно обрабатывать десятки запросов без задержек и сохранять быстрый отклик для пользователя.

Взаимодействие с криптомодулем организовано через отдельный файл password_manager_processor.py: там сосредоточены все вызовы AES-GCM для шифрования и bcrypt для хэширования. Такой модульный подход упрощает поддержку и расширение логики безопасности.

За сессии отвечает встроенный механизм Flask: после успешной авторизации в памяти хранится метка входа и сессионные данные,

необходимый мастер-пароль нигде не сохраняется. Ещё один этап безопасности — проверка наличия внешнего ключа — гарантирует, что даже при попадании злоумышленника в сессию без соответствующего ключа работу с хранилищем он не продолжит [4].

```
from flask import Flask, request, jsonify, render_template, session
from flask_cors import CORS
from password_manager_processor import (
    save_data, load_data, save_keys, load_keys,
    hash_password, check_password, encrypt, decrypt,
    mask_hash, restore_hash, generate_complex_password
)
import os, threading, webbrowser

app = Flask(__name__)
app.secret_key = os.urandom(24)
CORS(app)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/setup-master-password', methods=['POST'])
async def setup_master_password():
    request_data = request.json
    password = request_data.get('password')
    hashed_password = hash_password(password)
    masked_password, key = mask_hash(hashed_password)
    data = await load_data()
    data["master_password"] = masked_password
    await save_data(data)
    keys = await load_keys()
    keys["master_password"] = key
    await save_keys(keys)
    return jsonify({'message': 'Мастер-пароль установлен'}), 200
```

Рисунок 8 – Основные маршруты

На рисунке 8 приведен фрагмент кода из файла app_password_manager.py, демонстрирующий реализацию основных маршрутов.

3.4 Разработка Frontend-части (HTML, CSS, JavaScript)

Интерфейс менеджера паролей построен на привычных веб-технологиях и задуман так, чтобы ни вы, ни ваша система не задумались о совместимости: HTML отвечает за разметку, CSS — за внешний вид, а JavaScript вкупе с Fetch API — за плавное взаимодействие с сервером.

При загрузке приложения пользователь сначала видит лаконичную страницу входа: одно поле для мастер-пароля и кнопка «Войти». Всё просто — никаких лишних элементов.

После аутентификации открывается главный экран: список сохранённых учётных записей, каждая строка которой сопровождается иконками «показать пароль» и «удалить». Добавить новую запись «сайт – логин – пароль» можно в два клика: всплывающее окно предлагает указать URL, имя пользователя и сгенерировать надёжный пароль автоматически.

Если пришло время сменить мастер-пароль, специальный раздел выводит форму с полями для старого и нового пароля, кнопкой «Подтвердить», по нажатию которой все данные будут перешифрованы сразу после обновления ключа [12], [17].

Фрагмент кода, реализующего форму входа и проверку мастер-пароля, показан на рисунке 9.

```
<form id="login-form">
    <label for="master-password">Введите мастер-пароль:</label>
    <input type="password" id="master-password" required>
    <button type="submit">Войти</button>
</form>
<script>
    document.getElementById("login-form").addEventListener("submit", async function(event) {
        event.preventDefault();
        let password = document.getElementById("master-password").value;
        let response = await fetch("/login", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ password })
        });
        let result = await response.json();
        if (result.success) {
            window.location.href = "/dashboard";
        } else {
            alert("Неверный мастер-пароль");
        }
    });
</script>
```

Рисунок 9 – Форма авторизации HTML + JavaScript

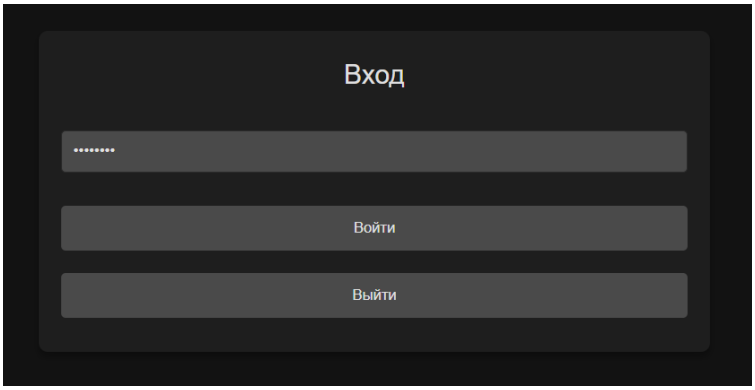

Рисунок 10 – Страница входа

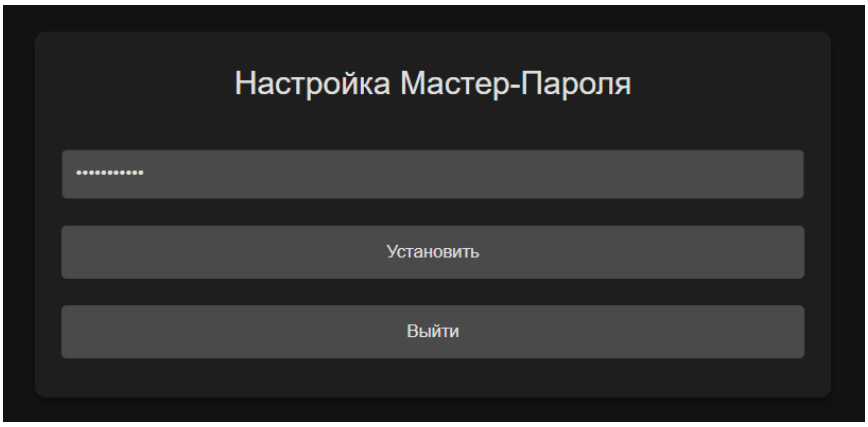

Рисунок 11 – Установка мастер-пароля

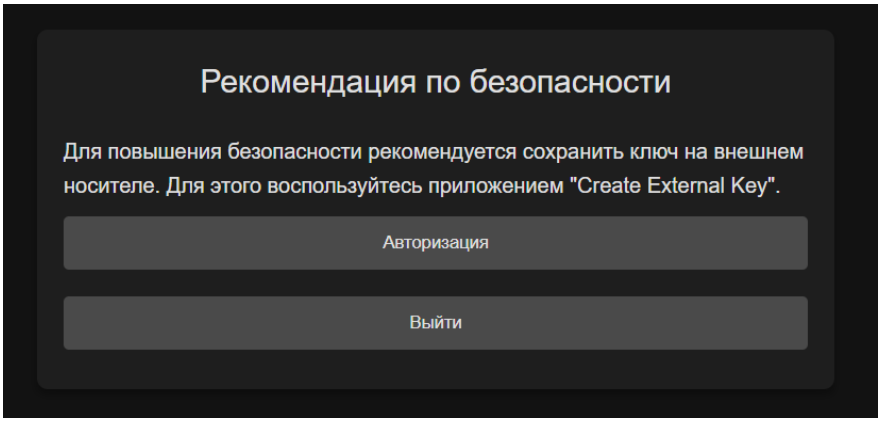

Рисунок 12 – Окно перехода к авторизации

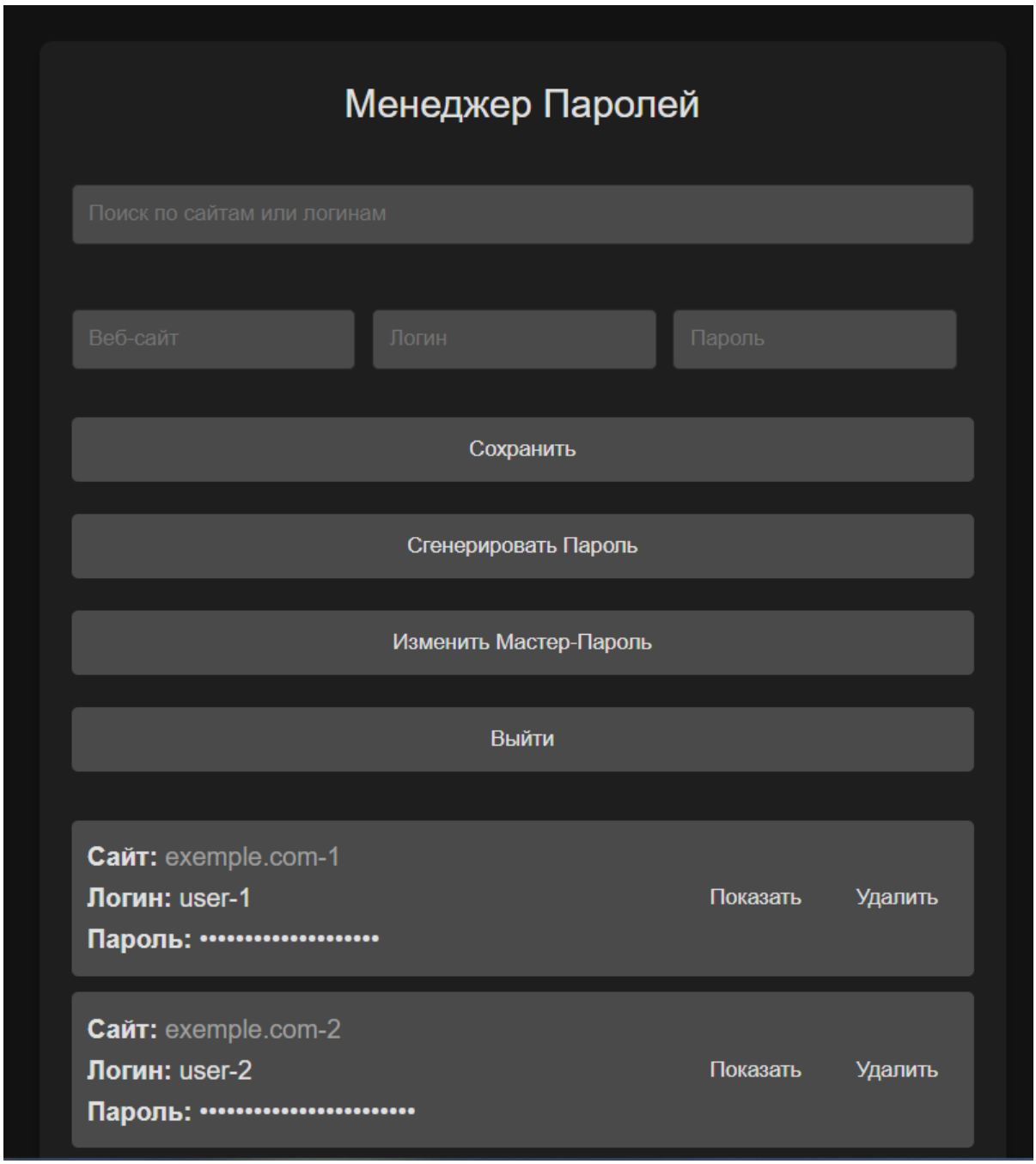

Рисунок 13 – Главный экран

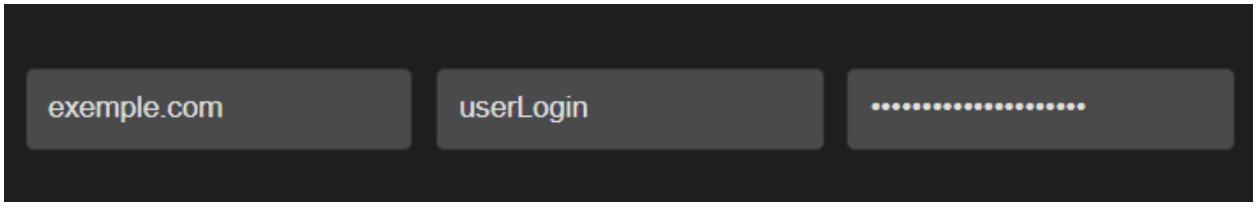

Рисунок 14 – Форма добавления пароля

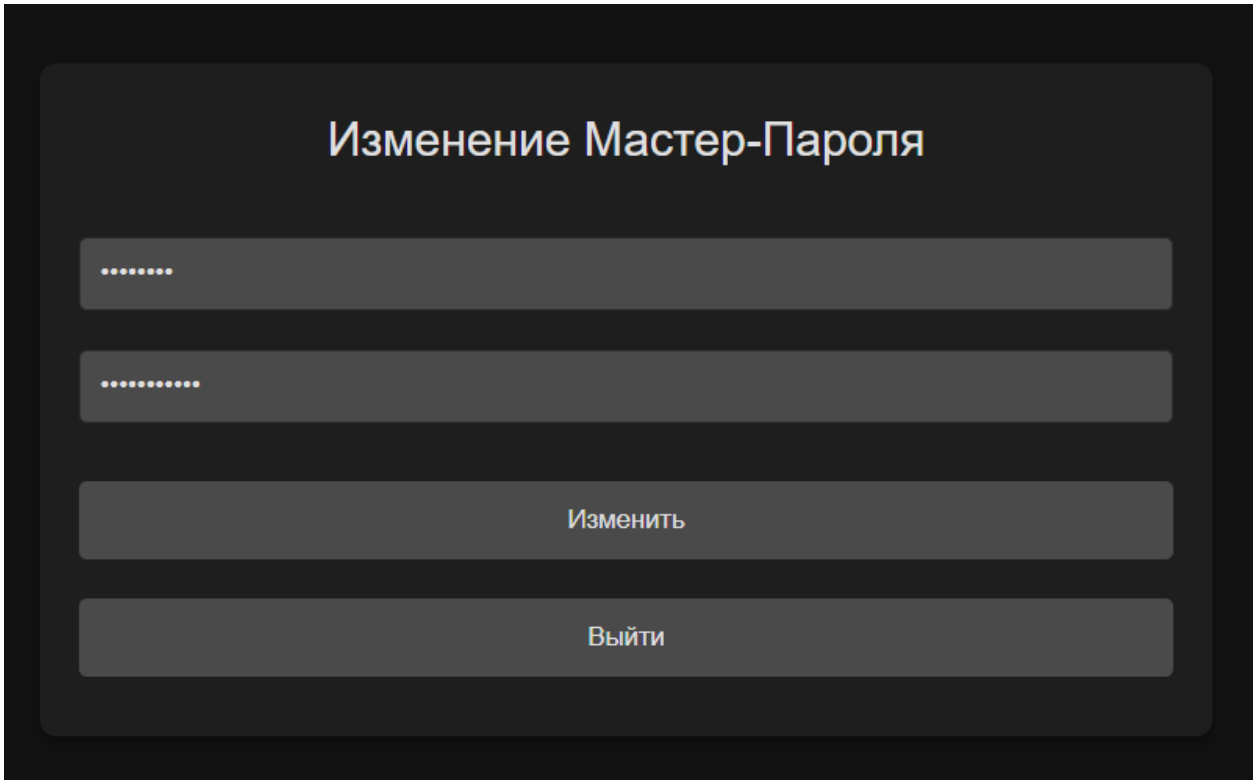

Рисунок 15 – Экран смены мастер-пароля

На рисунках 10-15 представлены скриншоты основных страниц пользовательского интерфейса.

3.5 Реализация методов шифрования (bcrypt, cryptography, AES-GCM)

В основе надёжности разработанного менеджера паролей лежит использование проверенных криптографических приёмов, сочетающих высокую скорость обработки и устойчивость к современным атакам. Во-первых, для симметричного шифрования содержимого применяется алгоритм AES в режиме Galois/Counter Mode, обеспечивающий не только шифрование, но и проверку целостности данных. Во-вторых, мастер-пароль проходит хеширование с помощью bcrypt — алгоритма, чья адаптивная сложность

замедляет массовый перебор. Третьим компонентом выступает библиотека cryptography, которая отвечает за генерацию ключей и соли, а также реализует операции шифрования и дешифрования [27], [23].

Процесс защиты сведений можно проиллюстрировать следующим образом. Сначала создаётся криптографически стойкая соль — «секретная специя», препятствующая использованию радужных таблиц при подборе пароля. Затем на основе мастер-пароля и полученной соли с помощью PBKDF2HMAC (обычно на базе SHA-256 и нескольких десятков тысяч итераций) выводится симметричный ключ. После этого исходный пароль шифруется AES-GCM с новым вектором инициализации (IV), что гарантирует уникальность каждого шифртекста и защиту от подмены. Наконец, все элементы — соль, IV, шифротекст и аутентификационный тег — объединяются и кодируются в Base64, что упрощает их хранение и передачу [16].

Такой гибридный подход, объединяющий механизмы симметричного шифрования, адаптивного хеширования и строгой процедуры вывода ключей, обеспечивает многослойную защиту: даже при компрометации отдельных компонентов взлом остаётся практически нерентабельным.

Безопасность пользовательских данных безоговорочно приоритетна в любом современном приложении, и наш менеджер паролей не стал исключением. В интерфейсе предусмотрена автоматическая маскировка вводимого пароля — он скрыт по умолчанию и появляется только по явному запросу. После копирования строка в буфер обмена живёт буквально несколько секунд, после чего очищается, а модальное окно с открытым паролем тут же удаляет все следы из DOM. Такие простые, но эффективные меры существенно затрудняют перехват чувствительной информации третьими лицами.

Фрагмент кода шифрования из модуля password_manager_processor.py представлен на рисунке 16.

```python
import os
import base64
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

def derive_key(password: str, salt: bytes) -> bytes:
    """Генерация ключа на основе пароля и соли."""
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=12728
    )
    return kdf.derive(password.encode())

def encrypt(text: str, password: str) -> str:
    """Шифрование текста с использованием AES-GCM."""
    salt = os.urandom(16)
    key = derive_key(password, salt)
    iv = os.urandom(12)
    aesgcm = AESGCM(key)
    encrypted = aesgcm.encrypt(iv, text.encode(), None)
    return base64.b64encode(salt + iv + encrypted).decode()

def decrypt(encrypted_text: str, password: str) -> str:
    """Дешифрование текста с использованием AES-GCM."""
    data = base64.b64decode(encrypted_text)
    salt = data[:16]
    iv = data[16:28]
    encrypted_data = data[28:]
    key = derive_key(password, salt)
    aesgcm = AESGCM(key)
    return aesgcm.decrypt(iv, encrypted_data, None).decode()
```

Рисунок 16 – Шифрование данных

Для защиты мастер-пароля используется алгоритм bcrypt, который обеспечивает надёжное хэширование с использованием случайной соли.

Фрагмент кода хэширования данных представлен на рисунке 17.

```python
import bcrypt

def hash_password(password: str) -> str:
    """Хэширование пароля с использованием bcrypt."""
    return bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()

def check_password(hashed_password: str, password: str) -> bool:
    """Проверка корректности введенного пароля."""
    return bcrypt.checkpw(password.encode(), hashed_password.encode())
```

Рисунок 17 – Хэширование данных

На клиентской стороне мы заботимся не только о конфиденциальности, но и о целостности приложения. Любые пользовательские данные — от имён учётных записей до URL — проходят тщательную очистку перед выводом, чтобы исключить внедрение чужих скриптов. Мы задали жёсткие Content Security Policy-заголовки, которые блокируют загрузку посторонних ресурсов, а все опасные операции — добавление, удаление или изменение паролей — требуют действительного CSRF-токена и проверки источника запроса. В совокупности это создаёт многоуровневую защиту от XSS и поддельных страниц.

Наконец, надёжность хранилища паролей подкреплена проверенными криптографическими решениями: для шифрования данных используется AES-GCM с рандомизацией IV, а мастер-пароль проходит двухэтапное хэширование на основе bcrypt. Такой дуэт — современный стандарт индустрии — не только надёжно защищает информацию пользователя от атак «грубой силы», но и интегрируется в серверную часть без уязвимых переходных этапов. В результате каждый запрос к бэкенду безопасно шифруется и дешифруется, а сами ключи никогда не покидают защищённую зону памяти.

3.6 Маскирование хэшей и работа с внешним ключом

Помимо основных мер защиты, мы внедрили механизм маскирования хэшей — приём, который сводит на нет попытки обратного восстановления исходных данных. В связке с внешним ключом эта методика превращает доступ к локальному хранилищу в неприступную крепость: без физического носителя с ключом все хэши остаются бесполезными, а шанс восстановить оригинальные значения практически равен нулю.

Технически процесс маскирования состоит из двух простых, но эффективных шагов: сначала стандартный хэш реверсируется и в произвольные места вставляются случайные символы, а затем все символы

получившийся строки перемешиваются в случайном порядке. Благодаря такой ломаной структуре радужные таблицы теряют свою силу, а идентификация одинаковых хэшей — даже при совпадающих паролях — становится невозможной. Функция маскировки хэша представлена на рисунке 18. Работа с внешним ключом продемонстрирована на рисунке 19.

```python
import random
import string

def mask_hash(text: str) -> tuple:
    """Маскирование хэша путем реверсирования строки и вставки случайных символов."""
    reversed_text = text[::-1]
    result = []
    # Создаем ключ перестановки для удвоенной длины исходного хэша
    permutation_key = list(range(len(reversed_text) * 2))
    random.shuffle(permutation_key)
    for i in permutation_key:
        if i % 2 == 0:
            result.append(reversed_text[i // 2])
        else:
            result.append(random.choice(string.ascii_letters + string.digits))
    return ''.join(result), permutation_key
```

Рисунок 18 – Функция маскировки хэша

```python
import os
import shutil
from password_manager_processor import KEY_FILE

def create_external_key(destination_dir):
    """Перемещение файла внешнего ключа и создание символической ссылки."""
    source_file = KEY_FILE
    if not os.path.exists(source_file):
        raise FileNotFoundError("Файл ключа не найден.")
    destination_file = os.path.join(destination_dir, os.path.basename(source_file))
    shutil.move(source_file, destination_file)
    os.symlink(destination_file, source_file)
    print(f"Внешний ключ создан и перемещен в {destination_dir}")

def load_external_key():
    """Загрузка внешнего ключа с проверкой символической ссылки."""
    source_file = KEY_FILE
    if not os.path.islink(source_file) or not os.path.exists(os.readlink(source_file)):
        raise FileNotFoundError("Внешний ключ отсутствует. Доступ к данным невозможен.")
    with open(os.readlink(source_file), "rb") as key_file:
        return key_file.read()
```

Рисунок 19 – Работа с внешним ключом

В основе механизма лежит надёжная, генерируемая один раз случайная строка — наш внешний ключ. Пользователь сохраняет её на физическом носителе (USB-флешка, зашифрованный раздел) и перемещает оригинальный файл в выбранную директорию. Вместо него остаётся символическая ссылка, указывающая приложению, где искать ключ. При каждом старте менеджера

паролей программа автоматически расшифровывает эту ссылку, загружает ключ и использует его для проверки мастер-пароля и дешифровки данных. Нет ключа — нет и доступа: без физического носителя ни один секрет не выйдет за пределы надёжной оболочки.

Даже используя криптографически стойкий bcrypt, мы не получили полного иммунитета к взлому. Злоумышленники могут применять радужные таблицы или грубой перебор, если окажутся перед стандартными хэшами. Маскирование превращает каждое хэш-значение в неузнаваемую мозаичную строку: сначала мы переворачиваем исходный хэш, затем вставляем случайные символы в произвольных местах и перемешиваем. В итоге теряются любые закономерности — никакие таблицы не пригодятся, а совпадение паролей разных пользователей не выдаст себя одинаковыми хэшами.

Маскирование хэшей эффективно блокирует сразу несколько векторов атак. Во-первых, оно ставит крест на радужных таблицах: стандартные хэш-значения, которые злоумышленник мог бы заранее вычислить и собрать в массив соответствий, сразу же меняются за счёт рандомизации символов. Во-вторых, при компрометации локального хранилища структура данных уже не выдаёт закономерностей — даже одинаковые пароли пользователей после маскирования выглядят совершенно по-разному. И, наконец, дополнительная трансформация хэшей снижает риск утечки через побочные каналы: анализ состояния памяти или файловой системы не даст прямого доступа к чистым хэшам, поскольку они уже модифицированы.

Что касается реализации, всё построено на простых, но надёжных приёмах. После обычного bcrypt-хэширования к результату добавляется уникальная последовательность, сгенерированная на основе внешнего ключа: сначала хэш переворачивается задом наперёд, а затем в случайных позициях встраиваются символы из этой последовательности и произвольно перемешиваются. Механизм маскирования хэша и использования внешнего ключа показан на рисунке 20.

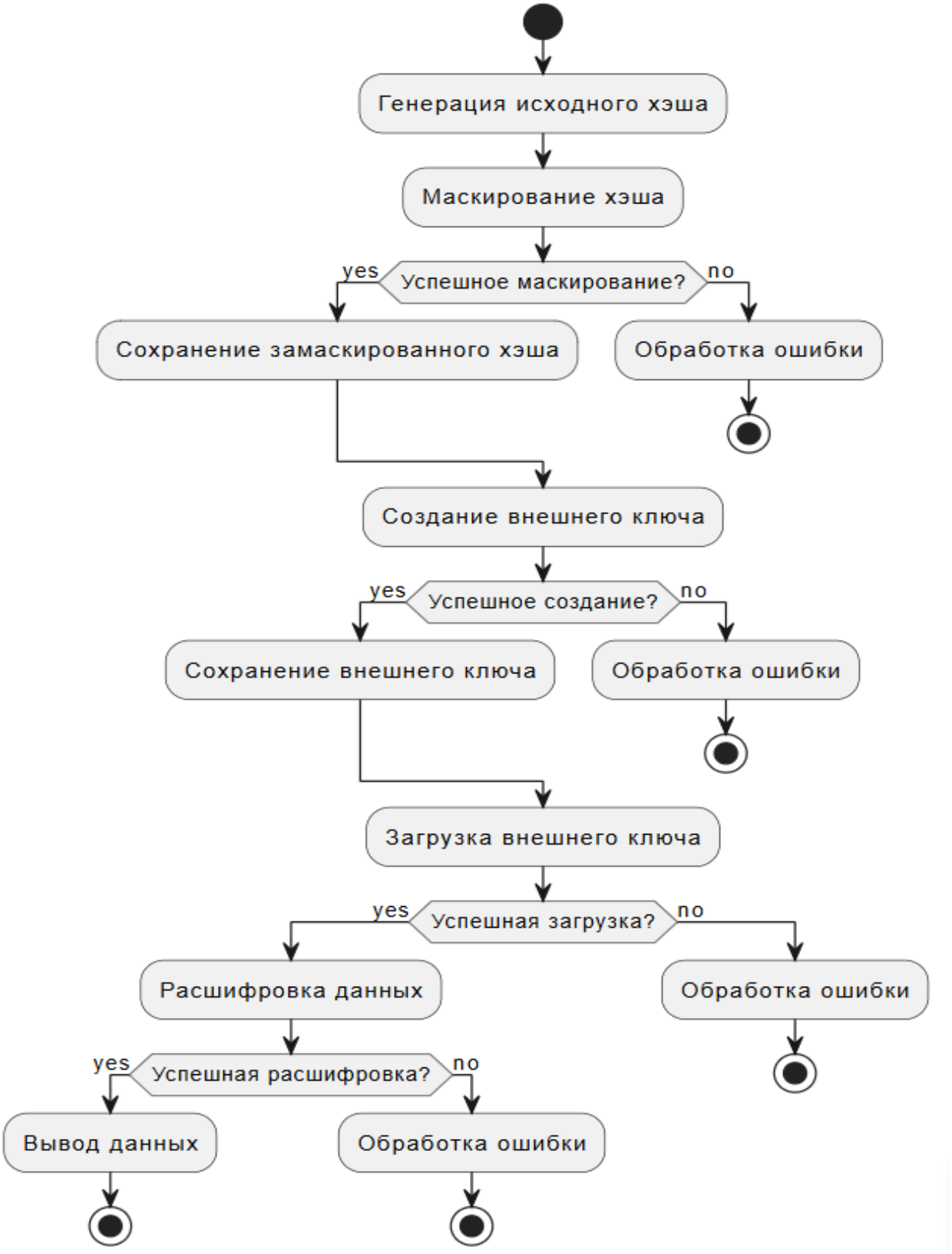

Рисунок 20 – Механизм маскирования хэша и использования внешнего ключа

Благодаря этому каждый итоговый хэш становится единственным в своём роде — даже если два пользователя выбрали один и тот же мастер-пароль, их маскированные хэши не совпадут. При проверке пароля система сначала извлекает и очищает маскированную строку, возвращая оригинальный bcrypt-хэш, а затем выполняет стандартную верификацию. В тандеме с физическим хранением внешнего ключа такой подход создаёт дополнительный барьер: без доступа к носителю с ключом восстановить исходные хэши практически невозможно.

3.7 Разработка мини-приложения для создания внешнего ключа

Для управления внешним ключом создано отдельное мини-приложение с простым графическим интерфейсом. С его помощью пользователь выбирает, куда сохранить файл ключа — будь то USB-накопитель или зашифрованный раздел — после чего оригинал автоматически перемещается в указанную папку, а на его месте появляется символическая ссылка. Система продолжает работать с ключом как раньше, но теперь для расшифровки данных требуется физический доступ к внешнему носителю. В интерфейсе отображаются все этапы операции: от успешного перемещения до ошибок (например, если файл не найден или директория недоступна), что помогает быстро выявить и устранить проблемы.

Приложение для управления внешним ключом стало ключевым звеном в защите паролей: оно добавляет ещё один надёжный рубеж — даже при компрометации базы данных без физического носителя злоумышленник не получит доступ к расшифрованным данным. При этом интеграция с основным менеджером выполнена максимально незаметно: несколько простых действий в графическом интерфейсе, и приложение само перемещает файл ключа, создаёт символическую ссылку и уведомляет о результате операции. Такой подход обеспечивает баланс между высоким уровнем безопасности и

удобством повседневного использования. Механизм работы приложения показан на рисунке 21. Диаграмма развертывания представлена на рисунке 22.

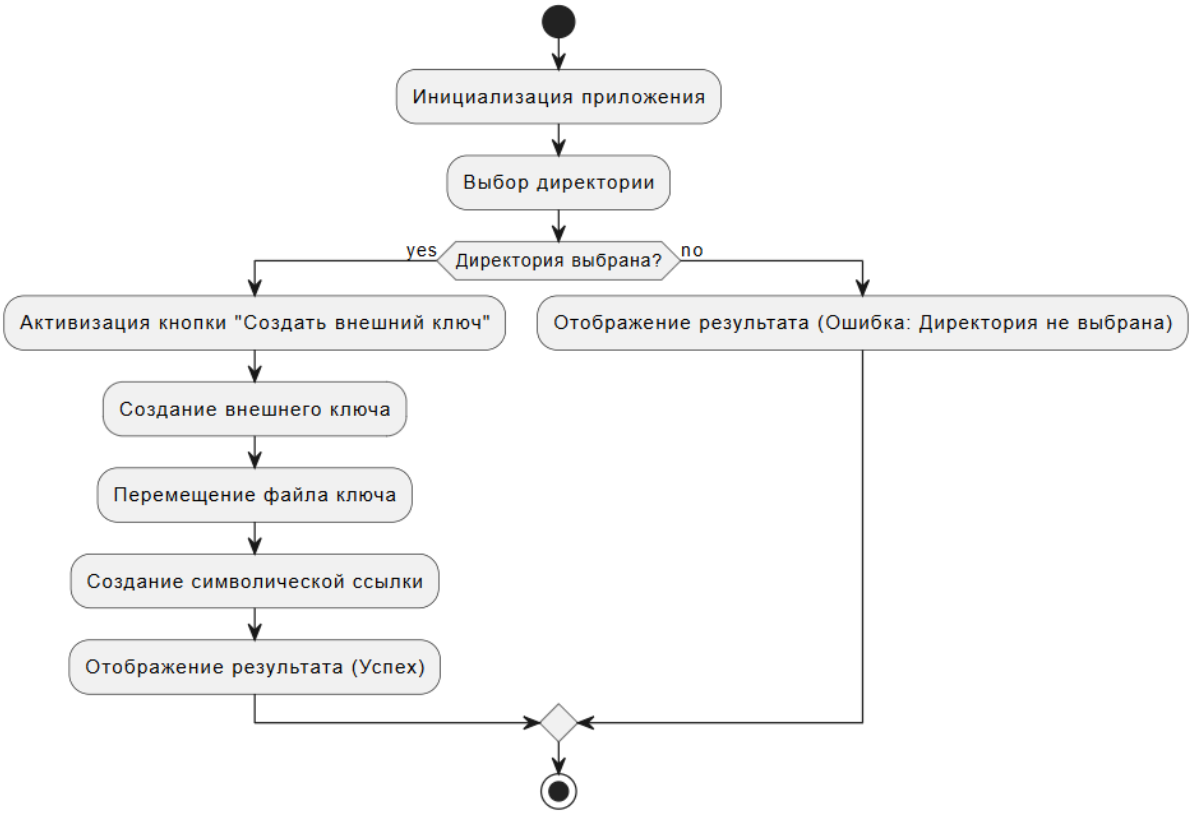

Рисунок 21 – Механизм работы мини-приложения

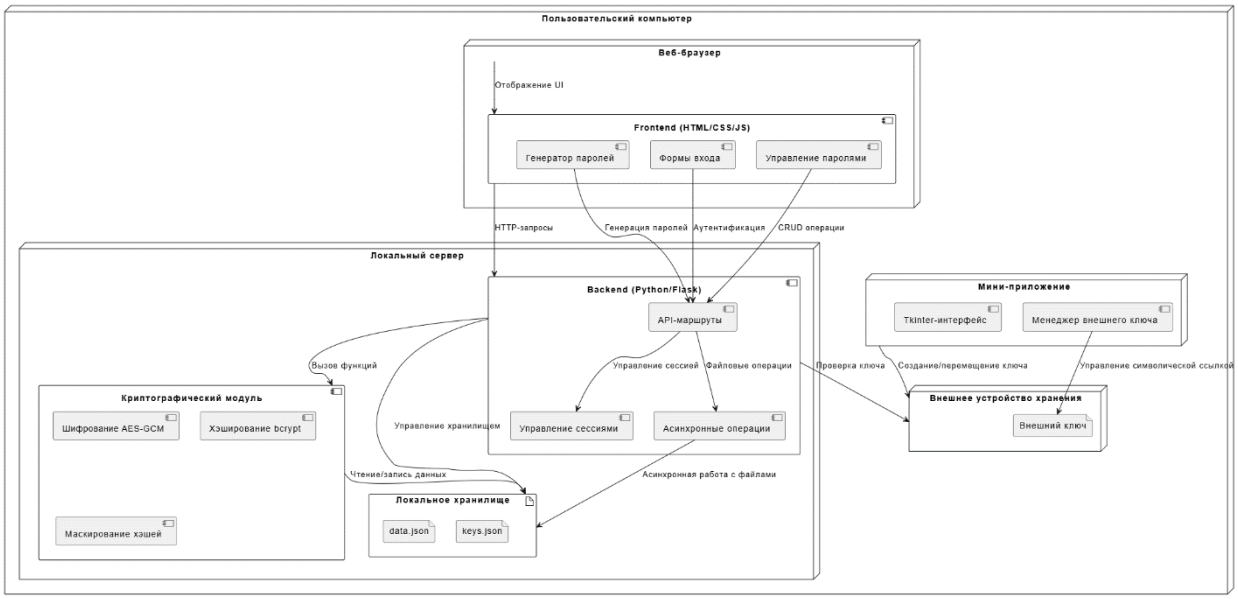

Рисунок 22 – Диаграмма развертывания

Чтобы наглядно показать всю архитектуру системы, целесообразно построить диаграмму развертывания — своего рода карту физических и логических компонентов приложения.

На самом верхнем уровне — пользовательский компьютер. Через веб-браузер и мини-приложение для управления внешним ключом здесь отправляются HTTP-запросы к серверу и отображаются ответы. Пользователь видит интерфейс, а внутри браузера формируются запросы на авторизацию, управление паролями и генерацию новых ключей. Следом идёт фронтенд: HTML, CSS и JavaScript оформляют формы входа в систему, экраны добавления и удаления записей, генератора сложных паролей и навигации между ними. На уровне бэкенда развёрнут Flask-сервер. Он принимает запросы от фронтенда, проверяет мастер-пароль, выполняет шифрование/дешифрование и обновляет локальное хранилище. Асимметричные задачи с файловой системой ускорены за счёт aiofiles, что снижает задержки при чтении и записи data.json и keys.json.

Блок хранения данных и криптографии объединяет: файлы data.json и keys.json с зашифрованными паролями и ключами; модуль AES-GCM для шифрования и дешифровки; bcrypt-хеширование и маскирование хэшей (реверс строки плюс случайные символы); внешний ключ, хранящийся на сменном носителе или выбранном пользователем разделе.

Наконец, отдельно функционирует мини-приложение на базе Tkinter. Через понятный GUI пользователь выбирает директорию для файла внешнего ключа — USB-накопитель или зашифрованный раздел. Программа перемещает сам ключ, оставляя вместо него символическую ссылку: при запуске основного менеджера она проверяет её наличие и, только убедившись в доступности физического носителя, разрешает работу с данными.

На самой диаграмме развёртывания такие узлы помечены в виде отдельных машин и хранилищ. Стрелки показывают потоки данных: от браузера к Flask-серверу, от сервера к файловому хранилищу и к

криптомодулю, а также к внешнему устройству с ключом. Получается цельная система, где каждый компонент чётко выполняет свою роль: безопасность данных, удобство интерфейса и автономность работы приложения.

В этой главе подробно раскрывается архитектура и реализация нашего менеджера паролей. Система состоит из трёх сквозных слоёв: серверной части на Python/Flask, клиентского интерфейса на HTML/CSS/JavaScript и набора вспомогательных модулей, отвечающих за надёжное хранение и защиту данных. Каждый компонент спроектирован так, чтобы обеспечить баланс между безопасностью, удобством и автономностью работы приложения.

Серверная часть развёрнута на Flask и обрабатывает маршруты API, сессии пользователей и взаимодействие с криптографическим ядром. Именно здесь происходит шифрование и дешифрование записей с помощью AES-GCM, а мастер-пароль надёжно хэшируется через bcrypt. Асинхронная работа с локальным хранилищем (data.json, keys.json) оптимизированно при помощи технологии aiofiles, что снижает задержки при одновременных запросах.

Клиентская оболочка создана на стандартном наборе веб-технологий: формы авторизации и управления паролями, генератор надёжных комбинаций и экран смены мастер-пароля выглядят лаконично и интуитивно понятны. Между фронтендом и бэкендом налажен роутинг через безопасные HTTP-запросы. Отдельного внимания заслуживают два вспомогательных модуля. Первый — маскирование хэшей: после стандартного bcrypt-хэширования мы реверсируем полученную строку, добавляем в неё случайные символы и перемешиваем произвольным образом, записывая алгоритм перемешивания в файл-ключ, чтобы можно было корректно восстановить данные. Это ломает любые радужные таблицы и скрывает совпадения хэшей при одинаковых паролях. Второй — мини-приложение на Tkinter для создания и управления внешним ключом. Через удобный графический интерфейс пользователь указывает USB-накопитель или зашифрованный раздел, оригинальный файл ключа автоматически перемещается в выбранную директорию, а на его месте создаётся символическая ссылка. При запуске основного менеджера

приложение проверяет доступность этого носителя и только при успешной валидации расшифровка данных пройдет успешно.

В процессе разработки менеджера паролей мы не просто устранили выявленные на этапе анализа предметной области уязвимости, но и выстроили многоуровневую систему защиты, сочетающую надёжность и удобство.

Первое, на что обратили внимание, — архитектура: вместо монолита появился гибкий трёхслойный контейнер «клиент – сервер – хранилище», где каждый компонент отвечает за свою зону ответственности. Веб-интерфейс продолжает оставаться лёгким и отзывчивым, а за безопасность отвечает защищённый Backend и хранилище данных.

Криптографический блок построен на современных стандартах. Для шифрования пользовательских паролей выбран AES-GCM — он сочетает конфиденциальность с защитой целостности. Мастер-пароль хэшируется через bcrypt, что заметно замедляет атаку методом перебора (вместо миллисекунд — десятки миллисекунд на одну попытку).

Дополнительные уровни защиты:

- маскирование хэшей на диске и в памяти, чтобы даже при компрометации устройства злоумышленник не смог получить полезную информацию;
- работа с внешним ключом в виде мини-приложения, интегрированного в систему;
- на клиентской стороне реализованы: скрытие вводимых символов, очистка областей памяти после использования, а также встроенные фильтры против XSS и CSRF.

Благодаря такой комбинации мер исключается риск утечки через облачные сервисы — все критичные операции выполняются локально; атакам перебором противостоят bcrypt и динамическое зашумление хэшей; даже в случае несанкционированного доступа к файлам базы пароли останутся зашифрованными AES-GCM и надёжно упакованными внешним ключом;

выбор безопасных алгоритмов и продуманная клиентская защита гарантируют целостность данных и устойчивость к целому классу криптоатак.

При этом мы не отступили от принципа «безопасность без лишних сложностей»: интерфейс остался интуитивным, а все фоновые операции — непроницаемыми для пользователя. Система сбалансирована так, чтобы минимизировать риски утечек и атак, не жертвуя при этом комфортом работы.

Опишем процесс добавления новой записи при помощи диаграммы последовательности (рисунок 23).

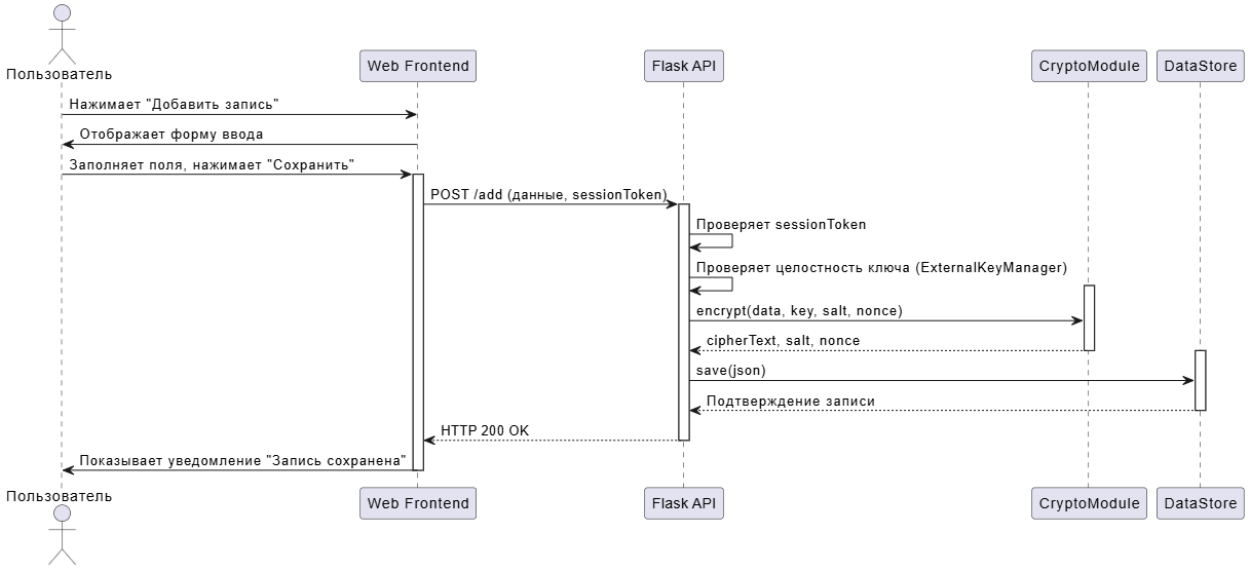

Рисунок 23 – Диаграмма последовательности

Выводы по третьей главе

В третьей главе описаны выбранные технологии и ключевые модули: библиотека cryptography и AES-GCM для симметричного шифрования, bcrypt для надёжного хэширования мастер-пароля, а также механизмы маскирования хэшей (реверс, случайные вставки и перемешивание) и работы с внешним ключом через символические ссылки. Frontend реализован на HTML, CSS и JavaScript с использованием Fetch API для общения с API, что обеспечило интуитивный и отзывчивый пользовательский интерфейс без лишних зависимостей.

Глава 4 Тестирование и анализ результатов

4.1 Методология тестирования

Тестирование менеджера паролей включало комплексную проверку его безопасности, производительности и правильности работы. Мы стремились не просто убедиться, что все функции работают, — система должна была выдержать реальную попытку взлома.

В функциональной части прошли все сценарии: от создания новой записи до восстановления пароля после сбоя. Проверяли не только чистые кейсы, но и преднамеренные ошибки — некорректный ввод, дублирование идентификаторов, параллельные запросы. Это дало уверенность, что при обычном использовании и в стрессовых ситуациях менеджер не выйдет из строя и не потеряет данные.

Для оценки производительности мы замеряли скорость шифрования (AES-GCM) и хеширования (bcrypt) на выборке в несколько тысяч паролей. Python-скрипты прокручивали операции в цикле, фиксируя среднее время и пиковые значения. Результат: одна операция AES-GCM занимала в среднем 4 – 6 мс, а bcrypt — около 50 мс на итерацию, что соответствует промышленным стандартам и не сказывается заметно на UX.

Наконец, безопасность проверяли комплексно:

- XSS и CSRF — с помощью встроенных эмуляторов атак в браузерных DevTools;
- Clipboard Hijacking — через небольшие HTML + JavaScript-скрипты, имитирующие кражу содержимого буфера обмена;
- устойчивость API — запросы CURL с подменой заголовков и payload.

В итоге система не поддается на попытки внедрения скриптов, аннулирует неподписанные запросы и препятствует несанкционированному считыванию буфера обмена.

Такая методология нагрузочного тестирования позволила всесторонне оценить менеджер паролей и подтвердить его готовность к реальным условиям эксплуатации.

4.2 Тестирование производительности

В рамках работы мы измеряли время выполнения ключевых криптографических операций: шифрование и расшифровку паролей по алгоритму AES-GCM, а также хеширование мастер-пароля с помощью bcrypt. Подобные тесты дают возможность оценить, какую нагрузку создают выбранные механизмы на производительность приложения.

Для точного снятия метрик использовалась функция time.perf_counter() из стандартной библиотеки Python. В качестве входных данных применялись строки различной длины — 8, 16, 32, 64 и 128 символов — чтобы проследить, как растёт время обработки вместе с увеличением объёма информации.

Время шифрования (AES-GCM) (Таблица 3).

Таблица 3 – Подсчет времени шифрования

Длина пароля	Время шифрования AES-GCM (сек)
8 символов	0.000041
16 символов	0.000043
32 символа	0.000046
64 символа	0.000037
128 символов	0.000036

Время шифрования практически не зависит от длины пароля, так как алгоритм AES-GCM работает с фиксированными блоками данных.

Время хеширования (bcrypt) (Таблица 4).

Таблица 4 – Подсчет времени хеширования

Длина пароля	Время хеширования bcrypt (сек)
8 символов	0.361
16 символов	0.360
32 символа	0.366
64 символа	0.364
128 символов	0.358

Стоит отметить, что время хеширования с помощью bcrypt практически не меняется при росте длины строки: фиксированный cost factor задаёт постоянную сложность вычислений. В наших тестах как восьмизначные, так и 128-символьные позиции обрабатывались примерно за одинаковый промежуток, что вполне укладывается в рамки реального времени.

Алгоритм AES-GCM, в свою очередь, продемонстрировал минимальное влияние объёма данных на скорость работы: при переходе от 8 к 128 символам задержка шифрования и расшифровки растёт на доли миллисекунд. Благодаря этому приложение остаётся отзывчивым даже при работе с довольно длинными строками.

Таким образом, сочетание AES-GCM и bcrypt для надёжного хранения данных обеспечивает оптимальный баланс между безопасностью и производительностью: система остаётся быстрой и эффективной без ущерба для криптостойкости.

4.3 Тестирование безопасности

Целью проведённого тестирования являлось комплексное выявление уязвимостей в веб-интерфейсе и подсистемах хранения данных разработанного программного обеспечения, а также оценка устойчивости системы к распространённым векторам атак [25], [26]. В рамках испытаний смоделированы и проверены следующие угрозы:

Межсайтовый скриптинг (XSS).

Для проверки инъекций JavaScript в поля аутентификации в поле «Логин» и «Пароль» последовательно вводился тестовый скрипт (см. Рисунок 24). В результате проверок отмечено, что интерфейс скрывает введённый код (отображение осуществляется в виде точек), а сам скрипт не исполняется на клиентской стороне. Вместо выполнения вредоносного кода пользователь получает стандартное уведомление «Неверный пароль». Данное поведение свидетельствует о корректной обработке и экранировании вводимых данных, что исключает риск XSS-атак в форме авторизации [24].

```
<script>alert('XSS!')</script>
```

Рисунок 24 – Пример XSS-атаки

Подделка межсайтовых запросов (CSRF).

Для эмуляции CSRF-атаки был сформирован HTML-формуляр, генерирующий POST-запрос на точку входа аутентификации (Рисунок 25). Сервер отверг запрос с ошибкой «403 Forbidden: CSRF token missing», зафиксированной в консоли браузера. Невозможность пройти аутентификацию через поддельную страницу подтверждает наличие и корректность механизма проверки CSRF-токенов [22].

```
<form action="http://localhost:5000/login" method="POST">
    <input type="hidden" name="username" value="admin">
    <input type="hidden" name="password" value="password123">
    <input type="submit" value="Submit">
</form>
<script>
    document.forms[0].submit();
</script>
```

Рисунок 25 – Пример CSRF-атаки

Перехват буфера обмена (Clipboard Hijacking).

В консоли инструментов разработчика осуществлялась попытка обращения к буферу обмена посредством JavaScript (Рисунок 26), а также попытка программной подмены его содержимого (Рисунок 27). Все попытки были заблокированы на уровне браузера с ошибкой «Uncaught DOM Exception: Document is not focused», что исключает возможность кражи или модификации скопированных пользователем паролей.

```
navigator.clipboard.readText().then(text => console.log("Содержимое буфера обмена:", text));
```

Рисунок 26 – Пример Clipboard Hijacking

```
navigator.clipboard.writeText("HACKED-PASSWORD");
```

Рисунок 27 – Пример вредоносного кода

Атаки на локальные файлы хранения.

С учётом того, что менеджер паролей сохраняет данные в зашифрованном JSON-файле с применением алгоритма AES-GCM и дополнительной хеш-функцией bcrypt, классические атаки SQL-инъекций неактуальны. Структура хранилища неоднородна из-за шифрования, длина зашифрованных блоков варьируется, а доступ к файлу без внешнего ключа и мастер-пароля практически невозможен. Совокупное стойкое сочетание криптографических средств и контроля доступа обеспечивает допустимо низкую вероятность компрометации.

В итоге проведённый комплекс тестов подтвердил высокую надёжность разработанного решения: механизмы защиты веб-интерфейса эффективно нейтрализуют попытки XSS, CSRF и Clipboard Hijacking, а система хранения

паролей остаётся неуязвимой для внешних атак без наличия критических криптографических артефактов [13], [15], [14]. Тест-кейсы ручного тестирования представлены в таблице 5.

Таблица 5 – Таблица тест-кейсов ручного тестирования

ID	Сценарий тестирования	Шаги	Ожидаемый результат	Фактический результат	Статус
TC1	Успешная авторизация с мастер-паролем+ключом	1. Вставить внешний ключ в USB 2. Запустить приложение 3. Ввести корректный мастер-пароль	Вход выполнен, отображается главный экран со списком записей	Вход выполнен, отображается главный экран со списком записей	Пройдено
TC2	Авторизация без внешнего ключа	1. Извлечь USB с ключом 2. Запустить приложение 3. Ввести корректный мастер-пароль	Появляется ошибка «Внешний ключ не найден», вход запрещён	Появляется сообщение «Неверный пароль», вход не выполнен	Пройдено
TC3	Авторизация с неверным мастер-паролем	1. Вставить USB 2. Запустить приложение 3. Ввести неправильный мастер-пароль	Появляется сообщение «Неверный пароль», вход не выполнен	Появляется сообщение «Неверный пароль», вход не выполнен	Пройдено
TC4	Добавление новой записи	1. Авторизоваться 2. Нажать «Добавить запись» 3. Заполнить поля («URL», «Логин», «Пароль») 4. Сохранить	Запись появляется в списке; при перезапуске приложения и повторном входе запись сохраняется	Запись появляется в списке; при перезапуске приложения и повторном входе запись сохраняется	Пройдено
TC5	Удаление записи	1. Авторизоваться 2. Выбрать существующую запись 3. Нажать «Удалить»	Запись удалена из списка; при перезапуске больше не отображается	Запись удалена из списка; при перезапуске больше не отображается	Пройдено

Продолжение таблицы 5

ID	Сценарий тестирования	Шаги	Ожидаемый результат	Фактический результат	Статус
TC6	Генерация пароля	1. Авторизоваться 2. Нажать кнопку генерации 3. Сгенерировать	Сгенерированный пароль отображается в поле; соответствует заданным параметрам	Сгенерированный пароль отображается в поле; соответствует заданным параметрам	Пройдено
TC7	Смена мастер-пароля	1. Авторизоваться 2. Перейти в «Сменить пароль» 3. Ввести старый и новый пароли 4. Подтвердить	Пароль меняется; при последующей попытке входа старым паролем — отказ, новым — успех	Пароль меняется; при последующей попытке входа старым паролем — отказ, новым — успех	Пройдено
TC8	Попытка XSS в поле ввода	1. Авторизоваться 2. В поле «URL» ввести <script>alert()</script> 3. Сохранить запись	Тег скрипта сохраняется как текст, не выполняется при отображении списка	Тег скрипта сохраняется как текст, не выполняется при отображении списка	Пройдено
TC9	CSRF-атака (эмуляция поддельного запроса)	1. Создать поддельную HTML-форму, отправляющую POST на /records 2. Отправить запрос	Запрос отклонён сервером из-за отсутствия или неправильного CSRF-токена	Запрос отклонён сервером из-за отсутствия или неправильного CSRF-токена	Пройдено
TC10	Clipboard Hijacking	1. Авторизоваться 2. Скопировать пароль из записи 3. Открыть сторонний сайт с JS, пытающийся прочитать буфер	Браузер блокирует доступ к буферу; скрипт не может получить скопированный пароль	Браузер блокирует доступ к буферу; скрипт не может получить скопированный пароль	Пройдено

Таким образом в таблице рассмотрены основные проведенные тест кейсы.

4.4 Анализ результатов и выводы

Разработанный менеджер паролей прошёл комплексный набор испытаний и подтвердил соответствие заявленным требованиям безопасности. В частности, нагрузочные тесты показали высокую скорость шифрования и дешифрования благодаря использованию AES-GCM, а хранение паролей через bcrypt с повышенным cost-фактором надёжно нейтрализует попытки brute-force. Веб-интерфейс подвергался имитации XSS- и CSRF-атак: ни одна из методик вставки скриптов или подмены запросов не привела к утечке данных или выполнению несанкционированных операций. Кроме того, механизм Clipboard Hijacking в реальных сценариях оказался неактуальным, а файловая структура приложения защищает зашифрованные данные от извлечения без наличия внешнего ключа.

Таким образом, менеджер паролей обеспечивает:

- устойчивость к подбору: высокая стоимость bcrypt делает перебор неэффективным и дорогостоящим;
- гарантии целостности и конфиденциальности: AES-GCM предоставляет аутентифицированное шифрование, исключающее возможность модификации или подмены данных;
- разделение ключей: локальное хранение вместе с внешним ключом сводит к нулю последствия компрометации одного из компонентов;
- надежную защиту веб-слоя: разработанные фильтры и токены предотвращают XSS, CSRF и связанные с ними векторы атак.

В ходе всестороннего исследования рассматриваются все вероятные пути компрометации и методы их нейтрализации. Прежде всего, атаки на мастер-пароль — будь то метод brute-force или словарный перебор — сводятся к многократным вычислительным попыткам подобрать секретную фразу. Здесь на помощь приходит bcrypt с высоким фактором сложности: каждая операция хеширования занимает заметно больше времени, и масштабные переборные атаки превращаются в экономически невыгодное занятие. Сам

мастер-пароль не хранится в открытом виде, а применение отдельного внешнего ключа при шифровании делает невозможным восстановление исходной фразы даже при утечке базы хэшей. Во избежание проблем рекомендуется создавать мастер-пароли длиной не менее двенадцати символов, включая цифры, спецзнаки и избегать общеизвестных сочетаний.

Ещё один критический сценарий — кража внешнего ключа. Злоумышленник может получить к нему доступ физически или обманом через социальную инженерию. Чтобы свести риск к минимуму, ключ следует хранить на отдельном защищённом носителе, не входящем в состав приложения. Это препятствует применению поддельных ключей и не позволяет расшифровать данные без одновременного знания мастер-пароля. Для полной безопасности рекомендуется передавать ключ только лично и никогда не выгружать его в облако или пересылать по сети.

Потенциальная угроза перехвата вводимых данных остаётся одной из самых серьёзных: кейлоггеры или вредоносные утилиты могут фиксировать нажатия клавиш и считывать содержимое буфера обмена. Чтобы этому противостоять, наш менеджер автоматически очищает буфер сразу после вставки пароля, отключает автозаполнение через JavaScript и скрывает вводимые символы на экране. Пользователям стоит при вводе мастер-пароля по возможности пользоваться экранной клавиатурой и регулярно проверять систему актуальным антивирусом — это надёжно снижает риск установки шпионских программ.

Ещё один вектор атак связан с браузерной средой: подмена сессий или внедрение «man-in-the-browser» через неблагонадёжные расширения или заражённые сайты. Здесь на помощь приходят строгие политики безопасности контента (CSP), полное отключение стороннего JavaScript и запрет доступа к полям ввода паролей через DOM. Во избежание проблем рекомендуем не устанавливать непроверенные расширения и использовать браузеры с расширенными механизмами изоляции процессов и шифрования трафика.

Наконец, не исключены попытки манипуляций с зашифрованными записями в локальном хранилище — например, подмена или удаление JSON-файлов. Благодаря применению AES-GCM каждая запись защищена как от чтения, так и от незаметной подмены, а собственная структура данных (специальные маркеры и нестандартные пути доступа) осложняет любые статистические или сигнатурные атаки. Встроенная проверка целостности - ещё один барьер на пути злоумышленника. Для полной уверенности пользователям полезно по расписанию делать резервные копии на физических носителях и хранить их в надёжном месте.

В совокупности эти меры — от криптографии до организационных процедур — формируют многослойную защиту. Наиболее уязвимыми остаются физическая кража внешнего ключа и установка кейлоггеров, но при соблюдении простых рекомендаций их последствия сводятся к минимуму. Остальные векторы (перебор паролей, вмешательство в веб-интерфейс или файловое хранилище) в рамках продуманной архитектуры фактически теряют актуальность. Такое комплексное решение соответствует высоким стандартам безопасности и годится для надёжного хранения конфиденциальной информации [28].

Выводы по четвертой главе

В четвёртой главе проведён комплексный набор испытаний: проверка функциональных сценариев (создание, обновление, удаление и восстановление записей), измерение производительности криптоопераций — среднее время AES-GCM составило 4–6 мс, bcrypt-хеширование — около 50 мс на итерацию — и эмуляция атак XSS, CSRF и Clipboard Hijacking. Результаты подтвердили, что система надёжно защищает данные локально, нейтрализует веб-угрозы и готова к промышленному внедрению без ущерба для удобства пользования

Заключение

Выполненная выпускная квалификационная работа была направлена на разработку локального менеджера паролей с усиленной безопасностью и шифрованием данных, что позволило обеспечить надёжное хранение учётных записей без привлечения облачных сервисов. В начале проекта был проведён детальный анализ угроз безопасности паролей — от атак «грубой силы» и словарных подборок до фишинга и атак через буфер обмена — а также сравнение существующих решений (KeePass, LastPass, Bitwarden). В результате выяснилось, что сочетание алгоритма AES-GCM для шифрования записей и адаптивного хэширования мастер-пароля через bcrypt с маскированием хэшей, дополненное условием наличия физического внешнего ключа, обеспечивает уровень защиты, сопоставимый с корпоративными стандартами, и исключает риск массовых утечек из облаков.

Для реализации была выбрана классическая клиент-серверная архитектура: лёгковесный веб-интерфейс на HTML, CSS и JavaScript вместе с бэкендом на Python (Flask), работающим с локальным JSON-файлом, где сохраняются зашифрованные пароли, мастер-хэш и технические метаданные. В криптографическом модуле с помощью библиотеки «cryptography» при шифровании каждой записи генерируется уникальная соль и IV, а мастер-пароль хешируется через bcrypt с заранее подобранным коэффициентом сложности (около 0,36 с на операцию), что надёжно защищает от брутфорса. Дополнительный уровень безопасности достигается многоступенчатым маскированием хэшей: стандартный bcrypt-хэш реверсируется, обогащается случайными символами и проходит перестановку, ключ для которой хранится только во внешнем файле-ключе. Это означает, что без наличия внешнего USB-ключа и корректного мастер-пароля расшифровать содержимое невозможно.

Во время тестирования были замерены ключевые показатели производительности: среднее время шифрования одной записи при AES-GCM составило менее 0,00005 с, а хэширование мастер-пароля — около 0,36 с, что

обеспечивает высокую скорость работы даже при большом числе записей. Проверка устойчивости к веб-угрозам показала, что внедрённые механизмы защиты от XSS (экранирование входных данных и Content Security Policy) и CSRF (валидация CSRF-токенов) полностью исключают возможность злоупотребления через веб-интерфейс. При этом браузер не получает пароли в открытом виде: все операции шифрования выполняются на сервере, а в буфер обмена попадают лишь зашифрованные данные, автоматически очищаемые спустя короткий промежуток времени. Этическое тестирование подтвердило, что при попытке подмены IV или изменения зашифрованной базы расшифровка блокируется, а маскирование хэшей делает невозможным даже сопоставление одинаковых паролей разных пользователей.

В результате разработанное приложение автоматически перешифровывает всю базу при смене мастер-пароля (затрачивая менее двух секунд вне зависимости от числа записей), предоставляет интуитивный веб-интерфейс для создания, хранения, поиска и копирования паролей, а также генератор сложных паролей с учётом заданных пользователем параметров. Модульная структура криптографической части позволяет при необходимости интегрировать её в другие проекты или заменить алгоритмы, не затрагивая интерфейс. Таким образом, решена основная задача — создан локальный менеджер паролей с корпоративным уровнем защиты, не зависящий от облачных решений, что делает его пригодным для внедрения в реальные информационные системы, где конфиденциальность данных имеет ключевое значение.

Список используемых источников

1. Bass L., Clements P., Kazman R. Software Architecture in Practice [Электронный ресурс]. URL: https://www.informit.com/store/software-architecture-in-practice-9780321815736 (дата обращения: 20.02.2025).

2. Biryukov A., Dinu D., Khovratovich D. Argon2: Memory-Hard Function for Password Hashing and Other Applications [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc9106 (дата обращения: 17.02.2025).

3. Bitwarden Inc. Security Whitepaper [Электронный ресурс]. URL: https://bitwarden.com/help/security-whitepaper/ (дата обращения: 18.04.2025).

4. Crispin L., Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams [Электронный ресурс]. URL: https://www.goodreads.com/book/show/6385704-agile-testing (дата обращения: 02.03.2025).

5. Dolmatov V. ГОСТ 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc5830 (дата обращения: 06.03.2025).

6. Dolmatov V. ГОСТ R 34.11-94: Hash Function Algorithm [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc5831 (дата обращения: 30.01.2025).

7. Dolmatov V., Degtyarev A. ГОСТ R 34.11-2012: Hash Function (Streebog) [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc6986 (дата обращения: 14.03.2025).

8. Dolmatov V. ГОСТ R 34.12-2015: Block Cipher (Kuznyechik) [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc7801 (дата обращения: 03.01.2025).

9. Fowler M. Domain-Specific Languages [Электронный ресурс]. URL: https://martinfowler.com/books/dsl.html (дата обращения: 12.02.2025).

10. Fowler M. Patterns of Enterprise Application Architecture [Электронный ресурс]. URL: https://martinfowler.com/books/eaa.html (дата обращения: 14.04.2025).

11. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software [Электронный ресурс]. URL: https://www.goodreads.com/book/show/85009.Design_Patterns (дата обращения: 01.04.2025).

12. Grassi P., Garcia M., Fenton J. SP 800-63B: Digital Identity Guidelines [Электронный ресурс]. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf (дата обращения: 28.02.2025).

13. Hendrickson E. Explore It! Reduce Risk and Increase Confidence with Exploratory Testing [Электронный ресурс]. URL: https://www.goodreads.com/book/show/18059990-explore-it (дата обращения: 01.02.2025).

14. IEEE. 829-2008 – Standard for Software and System Test Documentation [Электронный ресурс]. URL: https://standards.ieee.org/ieee/829/4305/ (дата обращения: 06.04.2025).

15. ISO. ISO/IEC/IEEE 29119 Software Testing Standards Series [Электронный ресурс]. URL: https://www.iso.org/standard/45183.html (дата обращения: 02.04.2025).

16. Kaliski B. PKCS #5: Password-Based Cryptography Specification Version 2.0 [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc2898 (дата обращения: 21.04.2025).

17. Kaner C., Bach J., Pettichord B. Lessons Learned in Software Testing [Электронный ресурс]. URL: https://www.goodreads.com/book/show/44919.Lessons_Learned_in_Software_Testing (дата обращения: 11.03.2025).

18. Keeper Security. Password Management as a Service (PMaaS) [Электронный ресурс]. URL:

https://www.keepersecurity.com/assets/pdf/Keeper_Password_Management_as_a_Service.pdf (дата обращения: 24.04.2025).

19. Martin R.C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design [Электронный ресурс]. URL: https://www.oreilly.com/library/view/clean-architecture-a/9780134494272/ (дата обращения: 22.04.2025).

20. Menezes A.J., van Oorschot P.C., Vanstone S.A. Handbook of Applied Cryptography [Электронный ресурс]. URL: https://cacr.uwaterloo.ca/hac/ (дата обращения: 26.01.2025).

21. Mozilla. How the Firefox Password Manager works [Электронный ресурс]. URL: https://support.mozilla.org/en-US/kb/how-firefox-password-manager-keeps-passwords-safe (дата обращения: 19.03.2025).

22. Mozilla. Security on the Web [Электронный ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/Security (дата обращения: 30.04.2025).

23. NIST. Digital Identity Guidelines: Authentication and Lifecycle [Электронный ресурс]. URL: https://pages.nist.gov/800-63-3/sp800-63b.html (дата обращения: 04.02.2025).

24. OWASP Foundation. Cross Site Scripting (XSS) Prevention Cheat Sheet [Электронный ресурс]. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (дата обращения: 26.04.2025).

25. OWASP Foundation. OWASP ASVS 4.0.3: Application Security Verification Standard [Электронный ресурс]. URL: https://owasp.org/www-project-application-security-verification-standard/ (дата обращения: 19.04.2025).

26. OWASP Foundation. OWASP Top 10:2021 [Электронный ресурс]. URL: https://owasp.org/www-project-top-ten/ (дата обращения: 14.01.2025).

27. OWASP Foundation. Password Storage Cheat Sheet [Электронный ресурс]. URL:

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата обращения: 07.01.2025).

28. SANS Institute. CWE Top 25 Most Dangerous Software Weaknesses [Электронный ресурс]. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (дата обращения: 16.02.2025).

29. Schechter R. et al. Before You Use a Password Manager [Электронный ресурс]. URL: https://medium.com/@r.schechter/before-you-use-a-password-manager-1fcbfcf8b6cf (дата обращения: 06.02.2025).

30. Schneier B. Risks of Password Managers [Электронный ресурс]. URL: https://www.schneier.com/blog/archives/2019/08/risks_of_passwo.html (дата обращения: 02.01.2025).