Санкт–Петербургский государственный университет

Грехнев Егор Олегович

Выпускная квалификационная работа

Автоматизация формирования транзакций и технологии хранения приватных ключей в блокчейне

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальные информатика и информационные технологии»
Основная образовательная программа СВ.5003.2016 «Программирование и информационные технологии»
Профиль «Автоматизация научных исследований»

Научный руководитель:
доцент, кафедра компьютерного моделирования и многопроцессорных систем, к.ф. - м.н.
Корхов Владимир Владиславович

Рецензент:
директор по развитию, ООО «Манзони», к.т.н.
Тихомиров Владимир Александрович

Санкт-Петербург
2020 г.

Содержание

Введение

В настоящее время блокчейн (распределенный реестр, данные в котором хранятся в блоках, создающих последовательную непрерывную цепочку) [1] является очень популярной технологией, а людей, использующих его, с каждым днем становится все больше и больше. Убедиться в этом можно посмотрев на график зависимости количества активных адресов в сети Ethereum от времени [2] (Рис. 1). В такой тенденции нет ничего удивительного, ведь распределенные реестры обладают огромным количеством преимуществ [3], основные из них: децентрализация, сохранность данных и прозрачность транзакций.

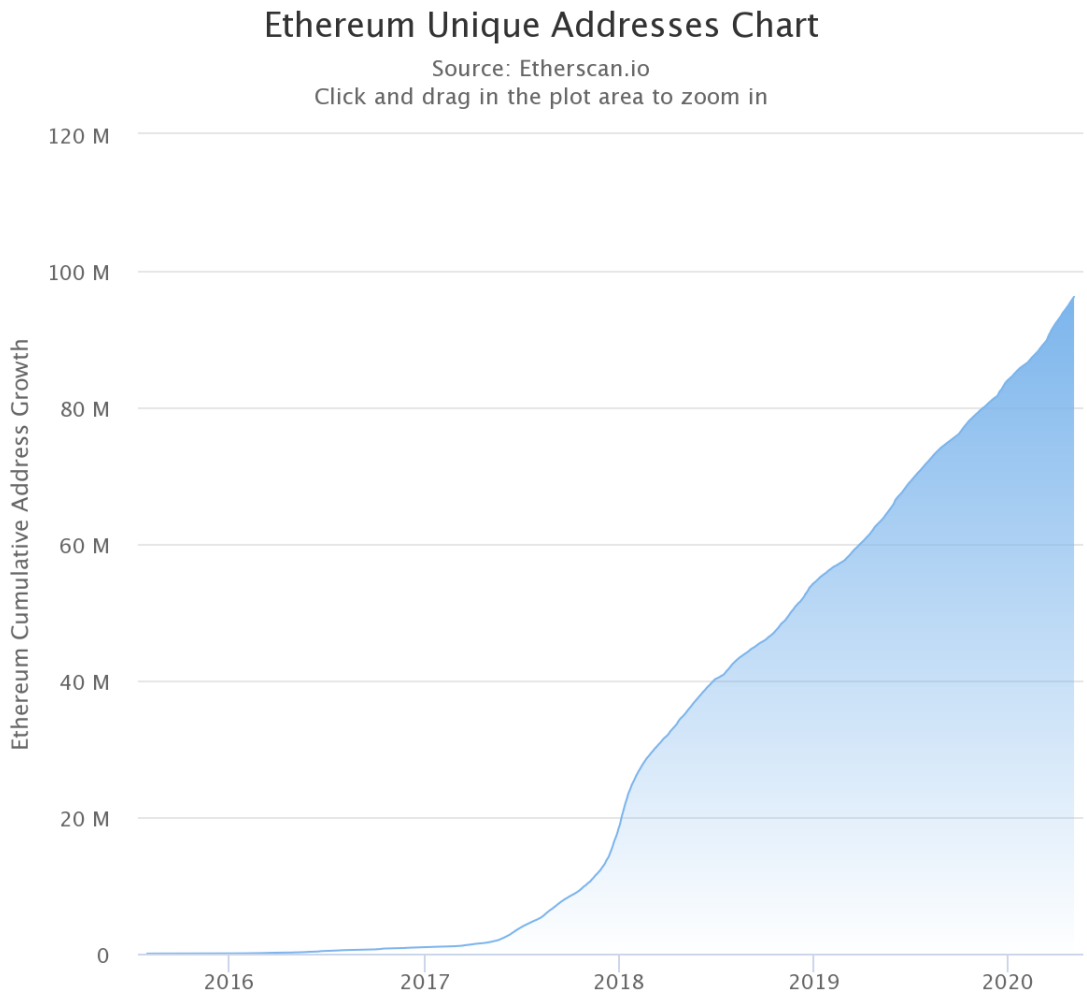

Рис. 1: Рост количества активных адресов в сети Ethereum.

Основанные на технологии блокчейн цифровые активы и криптова-

люты пользуются огромным спросом не только среди IT компаний, но и среди широкой массы пользователей. Транзакции, выполняемые в распределенных реестрах, а в особенности смарт-контракты (программы, хранящиеся и выполняющиеся в сети блокчейн) [4] могут предоставить людям огромные возможности, например, для развития бизнеса [5]. Транзакциями могут быть как простые переводы средств на счет, так и создание своих токенов для ICO (Initial coin offering) - форма привлечения инвестиций, которая заключается в продаже единиц токенов и является некоторой альтернативой классического IPO. Также оформление различных сделок, аналогов брачного договора или завещания, может ускорить процесс и исключить из дела посредников, что сэкономит людям время и деньги. В общем, блокчейн является крайне перспективной технологией, которая может улучшить и упростить жизнь людей.

Однако в такой системе тоже могут существовать свои недостатки. Необходимо выявить и решить некоторые из них.

Постановка задачи

В блокчейне используется технология цифровых подписей, основанная на сложных криптографических алгоритмах [6]. Приватный ключ – все, что требуется для подписи транзакции и гарантии ее неподдельности. Это очень удобно, однако без такого ключа невозможно получить доступ к своим средствам. Другими словами, проблемой является то, что потеря приватного ключа ведет к полной потере доступа к кошельку, а следовательно и ко всем средствам на нем. Так как приватный ключ невозможно восстановить, то это становится серьезной проблемой, что отталкивает многих пользователей от технологии.

Эту проблему теоретически можно решить логикой, реализованной в смарт-контракте, однако это приводит нас ко второй проблеме. Написание смарт-контрактов – задача сложная, требующая определенных прикладных знаний и навыков. К тому же, почти все блокчейны на данный момент не поддерживают классические популярные языки программирования и требуют изучения новых инструментов для этого (например Solidity [7]). Этот факт делает невозможным написание за приемлемое время необходимых программ-контрактов для людей, абсолютно не знакомых с этой областью.

Таким образом, задачей является нахождение и реализация такого решения, которое обезопасило бы владельцев токенов (цифровых активов) от потери доступа к своим средствам, автоматизируя создание и развертывание защищающего эти активы смарт-контракта. Также необходимо предоставить клиентам такой интерфейс, благодаря которому каждый технически неподготовленный пользователь смог бы воспользоваться текущим решением.

Для достижения этой цели необходимо выполнить следующие подзадачи:

- подробно изучить технологию блокчейн;
- проанализировать существующие решения, выявить их преимущества и недостатки;

- разобраться в структуре и принципах работы существующего сервиса компании по автоматизации определенных видов смарт-контрактов, предложившей решить выявленные проблемы в качестве выпускной квалификационной работы;
- основываясь на полученных знаниях предложить идею решения, выдвинуть технические требования, подобрать архитектуру проекта и инструменты для его реализации;
- реализовать минимально жизнеспособный продукт, удовлетворяющий поставленным требованиям;
- проанализировать полученное решение, сравнить с описанными существующими продуктами и подходами.

Обзор литературы

Статьи [1], [3] и [5] содержат доступное объяснение основных подходов и концепций технологии блокчейн. Публикации [4] и [6] дополняют вышеупомянутые работы подробным описанием идеи цифровых подписей, основанных на криптографических алгоритмах, и смарт-контрактов.

Ресурсы [12] и [13] являются существующими решениями, схожими с полученной идеей в плане интерфейса и реализации, а инструменты [8], [9], [10] и [11] расширяют понимание и представление идеи автоматизации формирования смарт-контрактов путем демонстрации альтернативных подходов.

Основная («White Paper» [27]) и техническая («Yellow Paper» [20]) спецификации сети Ethereum предоставляют минимальную информацию, необходимую для изучения и дальнейшей работы с платформой. Ресурс [28] является сайтом-реестром, на котором имеется вся информация о транзакциях в блокчейне, а также официальные статистические данные. Многие моменты, касающиеся оптимизации кода смарт-контрактов освещаются документацией основного языка программирования платформы [7].

Также имеется большое количество других ресурсов, ознакомление с которыми необходимо для выполнения поставленных задач. Так, идеи технологии хранения приватных ключей описаны в статье [29]. Подробные детали функции хэширования, используемой данной технологией можно найти в публикации [30]. Необходимая для интеграции сторонних решений (цифровых кошельков) в приложение информация доступно объясняется в официальных документациях этих продуктов (MetaMask [17], MyEtherWallet [18]), а актуальные данные по многим курсам криптовалют и цифровых активов доступна на крипто-бирже Binance [24].

Глава 1. Обзор существующих решений и подходов

1.1 Подходы по улучшению безопасности

Хранение средств на разных кошельках

Первым и самым очевидном решением является хранение средств на разных кошельках, другими словами, владение множеством различных приватных ключей. Этот подход позволяет распределить токены по разным местам. Такое решение значительно снижает вероятность потери всех средств, однако его использование крайне неудобно в связи с отсутствием центрального места управления активами.

Мультисиг контракты

Следующим популярным решением является хранение средств на специальном смарт-контракте. Мультисиг – это контракт, доступ к средствам которого имеют два или более адресов (при потере приватного ключа от одного адреса, можно управлять средствами с другого адреса). Такой подход включает в себя несколько недостатков. Во-первых, хранить средства на контракте не всегда надежно, так как небольшая ошибка в коде может не позволить вывести их оттуда [14]. Во-вторых, комиссия за операции вызовов функций перевода средств на смарт-контракте намного выше, чем обычная транзакция перевода с пользовательского адреса.

Сид фразы

Современные кошельки используют технологию так называемых сид фраз. Сид фраза – последовательность (размещение с повторениями) 12 случайных семантически не связанных слов из выбранного словаря (обычно 2048 слов). Имея такую последовательность кошелек позволяет создавать огромное количество приватных ключей, однозначно определяемых через сид фразу (все ключи можно восстановить из нее). Такую фразу из слов на английском языке проще запомнить, чем 256-значное число в виде приватного ключа, однако потеряв или забыв сид фразу человек точно так же теряет доступ ко всем средствам.

1.2 Продукты по автоматизации

Uml2Solidity [8]

Плагин для Eclipse (среда разработки). Позволяет строить UML диаграммы и генерировать из них каркас кода контракта на языке Solidity для платформы Ethereum. Для создания UML диаграмм используется Papyrus UML. Инструмент генерирует только оболочку смарт-контракта в виде сигнатур функций, код которых пользователь должен в дальнейшем реализовать самостоятельно. Использование сервиса может быть полезно разработчикам, уже имеющим знания в области технологии блокчейн и языка программирования Solidity, так как он предоставляет графический интерфейс для представления структуры и архитектуры будущего контракта. Таким образом, инструмент является узконаправленным и не упрощает работу с технологией широкой массе пользователей.

EtherScripter [9]

Веб-сервис, позволяющий создавать готовые смарт-контракты с помощью графического интерфейса. Инструмент позволяет строить смарт-контракты, используя синтаксические примитивы, представленные в виде графических фигур, которые сочетаются в интерфейсе перетаскивания (drag-and-drop). Однако использование синтаксических примитивов в графической форме все ещё требует знаний этих самых примитивов в языке программирования. Для технических пользователей EtherScripter не так удобен, как специализированная IDE, а для широкой массы пользователей он едва ли более понятен, чем текстовые редакторы.

OpenZeppelin [10]

Библиотека с открытым исходным кодом OpenZeppelin предоставляет набор небольших смарт-контрактов, которые могут быть использованы для достижения более сложного поведения благодаря возможности множественного наследования в языке Solidity. Для использования кода данной библиотеки в смарт-контрактах необходимо произвести его декомпозицию на множество более мелких контрактов, каждый из которых будет выполнять одну конкретную подзадачу. Это предоставляет возможность протестировать каждый смарт-контракт по отдельности, что несо-

мненно является плюсом в плане надежности и безопасности кода. Решение предлагает использовать контракты из библиотеки OpenZeppelin, которые представляют из себя готовые, протестированные, безопасные наработки, содержащие лучшие практики сообщества и регулярно проходящие аудиты. OpenZeppelin значительно упрощает создание смарт-контрактов для разработчиков, однако для его использования все еще необходимо хорошо разбираться в области. Тем не менее, это решение можно использовать в дальнейшем для упрощения и ускорения процесса написания кода смарт-контракта.

Unibright [11]

Unibright – универсальный фреймворк для интеграции технологии блокчейн с бизнесом и создания смарт-контрактов под его нужды. Разработчики продукта обещают создать платформу, которая решает схожие задачи с точки зрения автоматизации бизнес-процессов. Однако кроме описания идеи на данный момент в открытом доступе ничего нет.

Smartz [12] и Blockcat [13]

Smartz.io и Blockcat.io являются конкурирующими (по отношению к компании MyWish [15], предложившей решить описанные выше проблемы) сервисами, автоматизирующими формирование и развертывание смарт-контрактов на основе кастомизации готовых шаблонов. Имеют простой и удобный интерфейс, не требуют технических знаний и навыков для использования.

Глава 2. Описание проекта

2.1 Возможности сервиса

Описанная задача была поставлена компанией MyWish, которая занимается автоматизацией формирования и развертывания смарт-контрактов. Рассмотрим подробнее возможности, принципы работы и архитектуру существующего сервиса.

На данный момент сервис предоставляет возможность создания смарт-контрактов для ряда популярных блокчейн платформ: Ethereum, EOS, NEO, WAVES, TRON. Пользователи могут выбрать желаемый ими смарт-контракт, заполнить необходимые поля и за считанные минуты получить готовую программу, развернутую в необходимой им сети. К тому же, клиентам для каждого блокчейна предоставляется возможность попробовать создать контракт в тестовой сети (TESTNET) и бесплатно проверить функциональность их смарт-контракта. Были выявлены самые популярные и необходимые кейсы использования блокчейна, что привело к написанию конкретных шаблонов для смарт-контрактов на разных языках для каждой блокчейн платформы:

- token contract - генерация контракта для создания токена (единица ценности в блокчейне). Данная опция предполагает самостоятельное дальнейшее распространение этого токена в нужных клиенту целях;
- crowdsale contract (ICO) предполагает предыдущий пункт и автоматическое выставление токенов на продажу (первичное размещение монет);
- investment pool - контракт, для совместного накопления средств на нем среди многих пользователей для дальнейшего пользования, например инвестирования;
- airdrop - безвозмездное распределение токенов среди определенных адресов в виде подарка или ICO. Также примером может быть раздача токенов среди участников «крипто сообщества» за выполнение определенных действий;

- custom contract - при желании пользователь может сделать заказ на написание любого эксклюзивного смарт-контракта.

Сейчас компания стремится к тому, чтобы постепенно выделять особо популярные типы контрактов в отдельные сервисы. Примером может быть проект SWAPS.NETWORK [16], который предоставляет возможность составления смарт-контракта для обмена любыми токенами между пользователями. Так, планируется создание отдельного сервиса, решающего выявленные проблемы.

2.2 Архитектура проекта

Разные блокчейны поддерживают различные языки программирования для написания смарт-контрактов (Solidity, Go, Python и т. д.), что приводит к весьма трудоемкой работе. Для каждого типа контракта пишется шаблон, позволяющий впоследствии автоматически конфигурировать их код под введенные пользователем данные. Перед описанием взаимодействия всех компонентов проекта приведена их схема на Рис. 2.

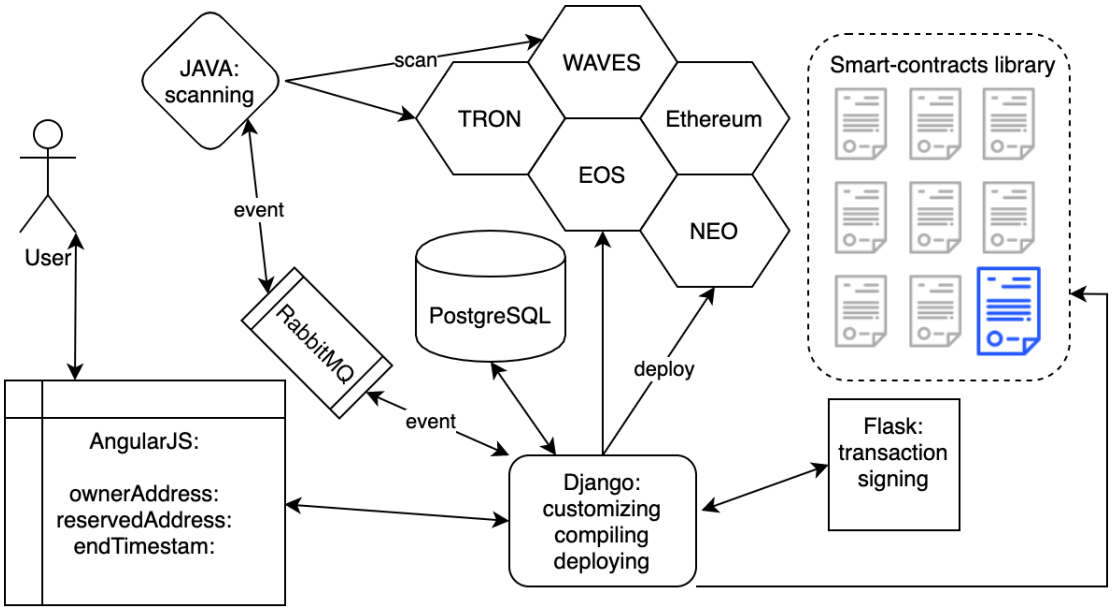

Рис. 2: Архитектура проекта MyWish.

После выбора пользователем блокчейн платформы, типа контракта и заполнения всех необходимых полей начинается работа на сервере, напи-

санном на Python. Эта работа включает в себя предварительную валидацию данных, встраивание пользовательских параметров, преобразование адресов и ключей в необходимый формат и компиляцию полученного контракта. Далее сохранение всех результатов в базу данных в целях безопасности.

После этого идет подключение к ноде (узлу) нужного нам блокчейна и создание объекта контракта в необходимом для него формате. Обычно это происходит через различные Python библиотеки, либо же работа идет напрямую с API конкретной платформы. Следующим шагом являются создание объекта для развертывания и цифровая подпись транзакции, заключающаяся в том, что она проходит с участием приватного ключа сервиса, что исключает необходимость отправлять по сети свои приватные ключи пользователям. После успешного подписания транзакция отправляется в блокчейн.

В блокчейне блоки могут формироваться некоторое длительное время в зависимости от конкретной сети: 30 секунд, минута, или например, как в сети Bitcoin, около 10 минут. Пока транзакция не попадет в блок нельзя утверждать пользователям, что все прошло успешно, поэтому, какой бы надежной не была технология блокчейн, задачей является гарантия достоверности и своевременное оповещение пользователей об успешности их операции, которые достигаются путем отслеживания транзакции до самого конца – попадания ее в конкретный блок, что говорит о записи в блокчейн (данное заявления не является достаточно точным, более подробно этот момент будет рассмотрен далее).

Для того, чтобы достоверно гарантировать пользователю факт успешного развертывания существует последний этап в процессе создания и публикации контракта. Приложение на Python записывает данные в базу и отправляет сообщение о транзакции в очередь (RabbitMQ). Это сообщение принимает «сканер», написанный на Java. Его задачей является сканирование всех реестров блокчейн платформ (они являются открытыми) до тех пор, пока не будет найдена искомая транзакция. В случае отсутствия транзакции в блоках, пользователь получает свои средства обратно.

Глава 3. Формирование концепции нового решения

3.1 Идея решения

На основе достоинств и недостатков существующих подходов была предложена идея смарт-контракта, основными свойствами и возможностями которого являются: контракт не хранит средства на своем счету и контракт способен перевести выбранные виды активов с основного адреса пользователя на заданный резервный в определенный день. Идею коротко можно описать таким сценарием:

- создание контракта через пользовательский интерфейс;
- настройка основных параметров контракта;
- потеря пользователем приватного ключа;
- перевод средств с основного адреса на резервный;
- возвращение токенов пользователю.

Графическое представление сценария защиты токенов можно увидеть на Рис. 3.

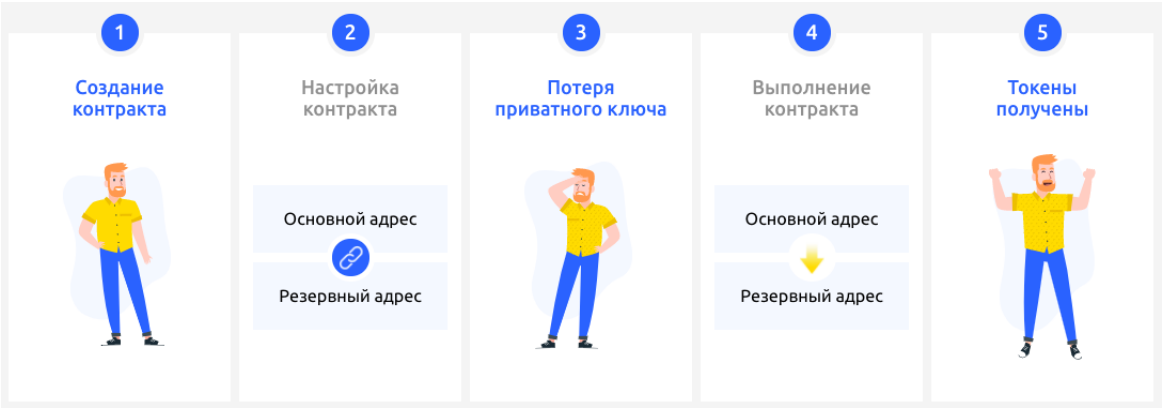

Рис. 3: Упрощенный сценарий процесса защиты токенов.

3.2 Выдвижение технических требований

Основываясь на предыдущих существующих решениях, имеющемся опыте компании по автоматизации различных типов контрактов и выдвинутой идее формируем следующие требования к функциональности сервиса и возможностям пользователя:

- пользователь указывает резервный адрес и дату вызова при создании контракта;
- смарт-контракт не должен иметь средств на своем счету;
- пользователь указывает список конкретных токенов, которые хочет защитить;
- контракт должен иметь доступ к переводу токенов в заданный день;
- пользователь полностью управляет своим контрактом – имеет доступ к вызову каждой функции;
- каждый пользовательский смарт-контракт должен иметь возможность выполниться независимо от состояния компании и сервиса в будущем;
- возможность отмены действия смарт-контракта пользователем в любой момент времени до момента вызова перевода средств;
- своевременное уведомление пользователей о скором вызове трансфера средств на резервный адрес;
- интеграция с популярными кошельками (Metamask [17], MyEtherWallet [18]) для подписания некоторых необходимых транзакций пользователем;
- возможность изменения набора выбранных для защиты токенов для перевода вплоть до момента вызова трансфера средств;
- возможность развертывания контрактов в тестовой сети с целью избежания комиссии;

- предоставление пользователю исходного кода смарт-контракта после его развертывания;
- полное покрытие кода смарт-контракта тестами.

3.3 Выбор конкретной сети

На данный момент существует большое количество распределенных реестров, поддерживающих реализацию и исполнение смарт-контрактов [19]. Одной из самых популярных таких платформ является Ethereum [20]. Технология имеет огромное количество публичных тестовых сетей, а также предоставляет удобные инструменты для развертывания своих локальных сетей, что может понадобиться при тестировании. К тому же, платформа поддерживает написание кода контрактов на языке Solidity, который является самым подробно документированным и популярным инструментом среди аналогов в этой и других блокчейн платформах. Все это привело к решению выбора данной платформы для первой реализации описанного решения.

Глава 4. Реализация смарт-контракта

4.1 Написание смарт-контракта

Одной из самых важных частей является непосредственно написание кода самого смарт-контракта.

В сети Ethereum есть несколько языков на которых можно написать смарт-контракт. Самыми популярными примерами могут быть Solidity - JavaScript подобный язык и Vyper - Python подобный язык. В настоящее время Solidity является наиболее хорошо документированным языком написания смарт-контрактов, к тому же он поддерживается самими разработчиками платформы Ethereum, поэтому выбор был сделан в его сторону.

Код смарт-контракта представляет собой класс, который имеет свои атрибуты: поля и методы. Как уже было сказано, основной функциональностью является возможность перевода выбранных пользователем токенов с основного адреса на резервный в определенные день. Учитывая это, контракт должен иметь следующие поля:

- ownerAddress – основной адрес пользователя, токены на котором требуется защитить;
- reserveAddress – резервный адрес, на который переводить токены;
- backendAddress – адрес самого сервиса, он необходим для возможности развертывания смарт-контракта без участия приватного ключа пользователя. Другими словами, не требуется предоставление приватного ключа пользователя в целях безопасности;
- endTimestamp – момент времени, после которого необходимо выполнить перевод.

Что касается функциональности смарт-контракта, то помимо основного метода перевода необходимо реализовать несколько дополнительных функций, перечислим их:

- constructor – инициализация смарт-контракта, присвоение переменным значений основного, резервного адресов, адреса сервиса и даты вызова;
- addTokenType – функция, принимающая и регистрирующая список токенов, которые необходимо будет защитить, на контракт;
- cancel – уничтожение (отмена) смарт-контракта;
- execute – функция перевода токенов, доступна только по наступлении даты вызова.

Работа смарт-контракта подразумевает обработку средств, имеющих реальную ценность в современном мире, это накладывает определенную ответственность перед пользователями, поэтому контракт должен быть основательно и качественно протестирован. Происходит это так: после кастомизации шаблона кода и компиляции контракта он развертывается в тестовую локальную сеть, где выполняются около 50 сценариев тестов различных транзакций. Только после успешного прохождения этого этапа контракт будет готов к настоящему развертыванию в тестовую или основную публичные сети.

4.2 Оптимизация кода смарт-контракта

Оптимизация кода смарт-контракта – очень важный и необходимый процесс. Это обусловлено двумя основными причинами:

- код смарт-контракта невозможно изменить после попадания транзакции блок, поэтому если он изначально написан не оптимально, то исправить это уже не получится;
- комиссия за транзакцию напрямую зависит от количества единиц операций и памяти, затрачиваемых на ее выполнение.

Рассмотрим подробнее второй пункт и попробуем разобраться в том, как можно снизить сумму комиссии за транзакции развертывания и управления смарт-контрактом.

Все транзакции в сети Ethereum выполняются на Ethereum Virtual Machine [21], в которой единицей вычислительной операции является «gas». Майнер - узел, который выбирает транзакции, выполняет их и формирует блок, тратит вычислительные ресурсы на выполнение этих действий, а следовательно на обработку и нашей транзакции. Отправитель транзакции должен компенсировать трату этих ресурсов, такая компенсация и является комиссией. Конкретная величина комиссии вычисляется как gas * gasPrice, где:

gas - количество газа, потраченного майнером при выполнении транзакции;

gasPrice - сумма, которую готов платить отправитель транзакции за единицу потраченного gas.

К тому же, майнерам выгодно выбирать транзакции с наибольшим gasPrice (так они могут получить наибольшее количество средств при наименьших вычислительных затратах), поэтому с маленьким значением этой величины ожидание за попадание транзакции в блок может быть значительно дольше обычного.

Таким образом, появляется задача – оптимизировать код так, чтобы транзакции за выполнение наших функций требовали как можно меньше газа. Сделав это, мы сможем увеличить gasPrice, обеспечив тем самым меньшее время ожидания попадания транзакции в блок и меньшее значение суммы комиссии, что будет полезным для пользователей.

Итак, код смарт-контракта можно оптимизировать как за счет количества операций, так и за счет используемой памяти. В нашем случае технически контракт не содержит каких-либо сложных алгоритмов, количество операций функций либо константно, либо линейно зависит от количества выбранных пользователем токенов. Также известно, что запись данных в хранилище является самой дорогостоящей операцией [22] виртуальной машины Ethereum, поэтому постараемся уменьшить количество требуемого газа за счет снижения потребления памяти, а именно за счет выбора наиболее подходящих типов данных хранения значений.

Поля ownerAddress, reserveAddress и backendAddress представляют собой строки, а endTimestamp – число. Для определения оптимальных ти-

пов данных необходимо провести опыт на простом смарт-контракте, включающем функцию-сеттер. Реализуем его (Рис. 4) и сделаем замеры количества потраченного газа транзакцией записи переменных в среде Remix IDE [23].

1) 0x519a3ed101CFff238cdFa6cb56DbDF61ef588050 - пример типичного адреса в сети Ethereum, попробуем записать его в переменные различных типов.

```
1  pragma solidity >=0.4.22 <0.7.0;
2
3  /**
4   * @title Storage
5   * @dev Store & retreive value in a variable
6   */
7  contract Storage {
8
9      string owner;
10
11     /**
12      * @dev Store value in variable
13      * @param num value to store
14      */
15     function store(string memory num) public {
16         owner = num;
17     }
18
19     /**
20      * @dev Return value
21      * @return value of 'number'
22      */
23     function retreive() public view returns (string memory){
24         return owner;
25     }
26 }
```

Рис. 4: Смарт-контракт записи значения в сторадж.

На Рис. 5 execution cost – количество потраченного газа на вызов функции, transaction cost – на всю транзакцию. По результатам эксперимента видно, что самым неэффективным для нашего случая в плане затрат газа является тип данных string (86463), далее идет bytes (44770), а лучше всего показал себя тип данных address (43759) – выгоднее использовать его. Однако тип данных address подходит только для записи адресов сети Ethereum, в случае необходимости записи строковой переменной другого вида будем использовать тип данных bytes.

Проведем такой же опыт для поля endTimestamp. Рассмотрим два типа данных: uint32 и uint. Первый представляет целое число от 0 до

Рис. 5: Пример успешно выполненной транзакции.

4294967295 и занимает 4 байта, второй же является синонимом uint256. Будем подавать на вход функций значение endTimestamp равное 1586957919.

Transaction cost в первом случае составляет 42757 газа, а во втором 41876. Этот на первый взгляд непредсказуемый результат обусловлен тем, что для хранения одной переменной типа uint32 на виртуальной машине сначала выполняется неявное его преобразование к типу uint256, только после чего происходит запись в сторадж. Этим и объясняется большее количество потраченного газа на запись в менее громоздкий по размеру тип данных. Проведем еще один эксперимент, посмотрим, каким будет результат при сохранении в сторадж не одной числовой переменной, а десяти. Пример кода такого смарт-контракта можно увидеть на Рис. 6.

Используя uint32, мы затратили всего лишь 77360 единиц газа, однако uint потребовал целых 222002. Таким образом, можно сделать вывод, что uint32 имеет смысл использовать при большом количестве числовых переменных, требуемых записи в сторадж. В данном же случае с одной переменной выгоднее использовать классический uint.

Сравним общее количество газа, которое можно потратить в описанных худшем и лучших случаях:

- запись трех строк в string и одного числа в uint32 - 302 146 газа;
- запись трех строк в address и одного числа в uint - 173 153 газа.

```
pragma solidity >=0.4.22 <0.7.0;

/**
 * @title Storage
 * @dev Store & retreive value in a variable
 */
contract Storage {

    uint32 endTimestamp;
    uint32 endTimestamp2;
    uint32 endTimestamp3;
    uint32 endTimestamp4;
    uint32 endTimestamp5;
    uint32 endTimestamp6;
    uint32 endTimestamp7;
    uint32 endTimestamp8;
    uint32 endTimestamp9;
    uint32 endTimestamp10;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint32 num) public {
        endTimestamp = num;
        endTimestamp2 = num;
        endTimestamp3 = num;
        endTimestamp4 = num;
        endTimestamp5 = num;
        endTimestamp6 = num;
        endTimestamp7 = num;
        endTimestamp8 = num;
        endTimestamp9 = num;
        endTimestamp10 = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retreive() public view returns (uint32){
        return endTimestamp;
    }
}
```

Рис. 6: Смарт-контракт для записи значения в 10 различных переменных.

Видно, что выбирая типы данных необдуманно, можно значительно увеличить комиссию за собственную транзакцию. На момент написания работы среднерыночное значение gasPrice составляло 15 GWEI = 0.000000015 ETH, а курс ETH в рублях имел значение 13 508,82 руб [24]. Учитывая эти данные, в худшем случае комиссия составила бы примерно 61 руб., в лучшем – 35 руб.

Рассмотрим еще один способ оптимизации смарт-контракта. Согласно официальной документации Solidity, поиск функции в контракте при ее вызове тоже затрачивает некоторое количество газа. Работает это так, что методы перебираются по очереди, и доступ к каждой последующей

функции в очереди затрачивает дополнительные 22 единицы газа. Помимо функций (методов), перебор идет и среди полей, поэтому с ростом их количества затраты могут сильно возрасти, к тому же транзакции вызовов функций, в отличие от единичной транзакции развертывания смарт-контракта, могут вызываться повторно огромное количество раз.

Необходимо разобраться с тем, по какому именно правилу упорядочиваются функции и переменные. В официальной документации сказано, что они сортируются согласно их «Method ID», который является первыми 4 байтами результата выполнения хэш функции keccak-256 [25] от сигнатуры объекта (метода или поля класса). Сигнатура метода – его название вместе с типами входных параметров (например, execute(address,uint)). Таким образом, видно, что можно подобрать название необходимой нам функции так, чтобы ее «Method ID» оказался минимальным. Это в свою очередь приведет к тому, что функция будет располагаться первой в очереди и не будет затрачивать дополнительные единицы газа на ее поиск.

Учитывая это, был придуман алгоритм, который подбирает название так, чтобы новая сигнатура удовлетворяла вышеописанным требованиям и сохраняла в себе исходное название функции. Результатом работы программы является имя функции, полученное конкатенацией старого названия с некоторым набором символов. Набор символов подбирается из цифр и латинских символов с постепенно увеличивающейся длиной до тех пор, пока не будет выполнено требуемое условие новой сигнатуры. Код реализованного алгоритма на языке Python можно увидеть на Рис. 7.

Такая оптимизация может быть актуальна, когда имеется один или несколько методов, которые вызываются часто относительно остальных. В нашем же случае, единственной функцией, которая будет вызываться много раз является «addTokenType» (пользователь может сколько угодно раз добавлять список новых токенов для защиты, поэтому количество ее вызовов неограниченно), ее сигнатура – «addTokenType(address[])». Запустив скрипт, получим новое оптимальное название функции - «addTokenType_cy», «Method ID» которой равен нулю, а приоритет при поиске которой будет максимальным.

Таким образом, видно, что можно значительно снизить стоимость

```python
import itertools
import string

from web3 import Web3


def find_new_signature(func_signature):
    char_sequence = string.digits + string.ascii_letters
    for permutations_length in range(1, len(char_sequence)):
        permutations = itertools.permutations(char_sequence, permutations_length)
        for permutation in permutations:
            new_func_signature = func_signature.replace('(', '_' + ''.join(permutation) + '(')
            if Web3.keccak(text=new_func_signature).hex().startswith('0x0000'):
                return new_func_signature

if __name__ == '__main__':
    func_signature = input()
    print(find_new_signature(func_signature))

```

Рис. 7: Скрипт подбора имени функции с нулевым «Method ID».

комиссии за транзакции в сети Ethereum за счет оптимизации кода. С учетом всех этих исследований был реализован смарт-контракт, код которого можно увидеть в репозитории на GitHub [26].

Глава 5. Реализация сервиса автоматизации

Архитектура решения в целом осталась такой, которая была изображена на Рис. 2. Рассмотрим подробнее некоторые отдельные компоненты проекта, после чего будет описан полноценный сценарий работы продукта.

5.1 Пользовательский интерфейс

Напомним, что одной из важнейших целей работы является создание простого и удобного пользовательского интерфейса, который позволит легко создавать и управлять смарт-контрактами. Для осуществления поставленных целей было решено выбрать популярный JavaScript фреймворк, а именно AngularJS.

Основной функционал фронтенда можно перечислить тезисно:

- регистрация, авторизация пользователя в приложении, а также авторизация через сторонние сервисы;
- интерфейс создания смарт-контракта – выбора даты вызова и резервного адреса;
- предоставление платежной системы, поддерживающие разные валюты;
- интеграция с кошельком MetaMask для удобства подписания необходимых транзакций пользователем;
- предоставление возможности выбора конкретных токенов для защиты;
- интерфейс функции отмены смарт-контракта;
- возможность просмотра всех данных и исходного кода контракта.

Интересным моментом является интеграция с кошельком MetaMask для подписания транзакций пользователем. Как уже было сказано, смарт-контракт развертывается непосредственно сервисом. Это делается для того, чтобы не требовать от пользователя его приватного ключа в целях

безопасности. Однако пользовательский токен – самостоятельный смарт-контракт, где прописаны адреса, которые могут управлять конкретным количеством его единиц. Поэтому чтобы развернутые смарт-контракты смогли перевести выбранный набор токенов в определенный момент времени, им необходимо иметь доступ к ним. Такой доступ может дать только владелец токенов, оформив специальную транзакцию от своего лица.

Оформление таких транзакций вручную крайне сложная задача, особенно для широкой массы пользователей, поэтому было решено интегрировать в сервис такое решение, как MetaMask. MetaMask – браузерная утилита, предоставляющая все необходимые функции кошелька. Также приложение позволяет создать новый приватный ключ (по сути кошелек), импортировать уже имеющийся, а главное, что эти данные хранятся локально и не передаются по сети. Кошелек имеет API для разработчиков, что позволяет предоставить пользователю легкий и удобный способ безопасного подписания предоставленных ему сервисом необходимых транзакций на месте, без необходимости отправки своего ключа по сети. Именно это и было реализовано для возможности предоставления доступа созданному смарт-контракту к выбранным токенам.

5.2 Сервер приложения

Для написания сервера необходимы инструменты, которые предоставляют средства для удобной работы с блокчейном Ethereum. Таким средством является библиотека web3, реализация которой имеется в таких языках, как Python, JavaScript и Java. Основной сервис компании был написан на Python, поэтому выбор был сделан в сторону этого языка. Конкретно для написания бэкенд сервисов были использованы наиболее популярные фреймворки – Django и Flask.

Основной функционал сервера приложения можно перечислить тезисно:

- авторизация, регистрация, создание сущностей смарт-контрактов;
- реализация платежной системы в разных валютах;

- кастомизация смарт-контрактов, основанная на пользовательских данных;
- компиляция полученных индивидуальных контрактов;
- тестирование скомпилированного смарт-контракта;
- развертывание смарт-контракта в выбранной сети Ethereum;
- оповещение пользователей почтовой рассылкой за неделю и за сутки до времени наступления исполнения смарт-контракта;
- возможность управление созданным смарт-контрактом.

Весь сервер приложения был разделен на три подсервиса: основной сервис (регистрация и авторизация пользователя, создание и управление смарт-контрактом), сервис отслеживания и регистрации платежей, сервис подписания транзакций.

Основной сервис было решено реализовать на Django, так как этот инструмент имеет огромное количество встроенных модулей, позволяющих значительно ускорить процесс разработки личных кабинетов, панели администратора и т. п..

Оставшиеся два сервиса значительно проще по своей логике и не требуют такого количества высокоуровневых возможностей, поэтому их было решено реализовать, используя фреймворк Flask. Доступ к сервису подписи транзакций будет иметь ограниченное количество человек в компании, то есть приватные ключи, которыми подписываются транзакции развертывания пользовательских смарт-контрактов и вызовов некоторых других функций находятся в безопасности, риск их утечки минимален.

5.3 Сканер реестров

Блокчейн - распределенная сеть, в которой узлы, работающие независимо друг от друга, пытаются формировать блоки. Возможна такая ситуация, когда один узел выпустил блок, отправил его остальным участникам сети, но блок еще не успел дойти до всех, как другой узел сделал

свой собственный блок – получается разветвление сети. Возникает вопрос, какой блок считать правильным? Для этого в сети Ethereum существует технология GHOST [27], результатом работы которой остается только одна истинная ветка истории блоков для каждого узла. Так, каждый узел выбирает наиболее «тяжелую» ветку блоков, определенную данным подходом. Остальные ветки отбрасываются, а все транзакции, попавшие в эти ветки не попадают в блокчейн. В сети Ethereum блоки формируются относительно часто – в среднем раз в 10 секунд, поэтому возникновение явления разветвления сети не является редкостью. Таким образом, даже при попадании нашей транзакции в блок мы не можем быть полностью убеждены в том, что транзакция безвозвратно попала в сеть, а следовательно в ее выполнении.

Для решение этой проблемы был реализован так называемый сканер, задача которого заключается в сканировании открытого реестра etherscan [28] для получении информации о количестве блоков, подтверждающих наш. Разработчиками сети Ethereum эмпирически было подобрано число 5 – максимальная длина разветвленных цепочек блоков, при которой сеть еще считается стабильной. Поэтому для гарантированного подтверждения попадания транзакции в блокчейн в стабильной сети сканер был настроен на поиск цепочки длиной в 6 блоков, подтверждающей транзакцию, созданную сервисом. После подтверждения необходимым количеством блоков сканер генерирует событие, которое отправляется на сервер приложения с помощью брокера сообщений RabbitMQ.

Ниже можно увидеть список транзакций, подтверждения которых ожидает сканер:

- развертывание смарт-контракта;
- осуществление платежа – пополнения баланса;
- выбор токенов пользователем;
- добавление списка токенов на контракт;
- отмена действия смарт-контракта;

- перевод средств смарт-контрактом.

5.4 Технология хранения приватных ключей

Так как сервис является коммерческим проектом, то необходимо реализовать инструмент оплаты предоставляемых услуг. Однако существует некоторые условия, которые делают эту задачу не совсем тривиальной. Есть потребность в реализации возможности, при которой пользователь мог бы оплачивать услуги с любого удобного ему в данный момент адреса любой популярной криптовалютой или токеном. Это ведет к проблеме сложности определения от кого конкретно пришел платеж. Описанную проблему может решить технология HD ("hierarchical deterministic") Wallet [29], концепция которой схематично изображена на Рис. 8.

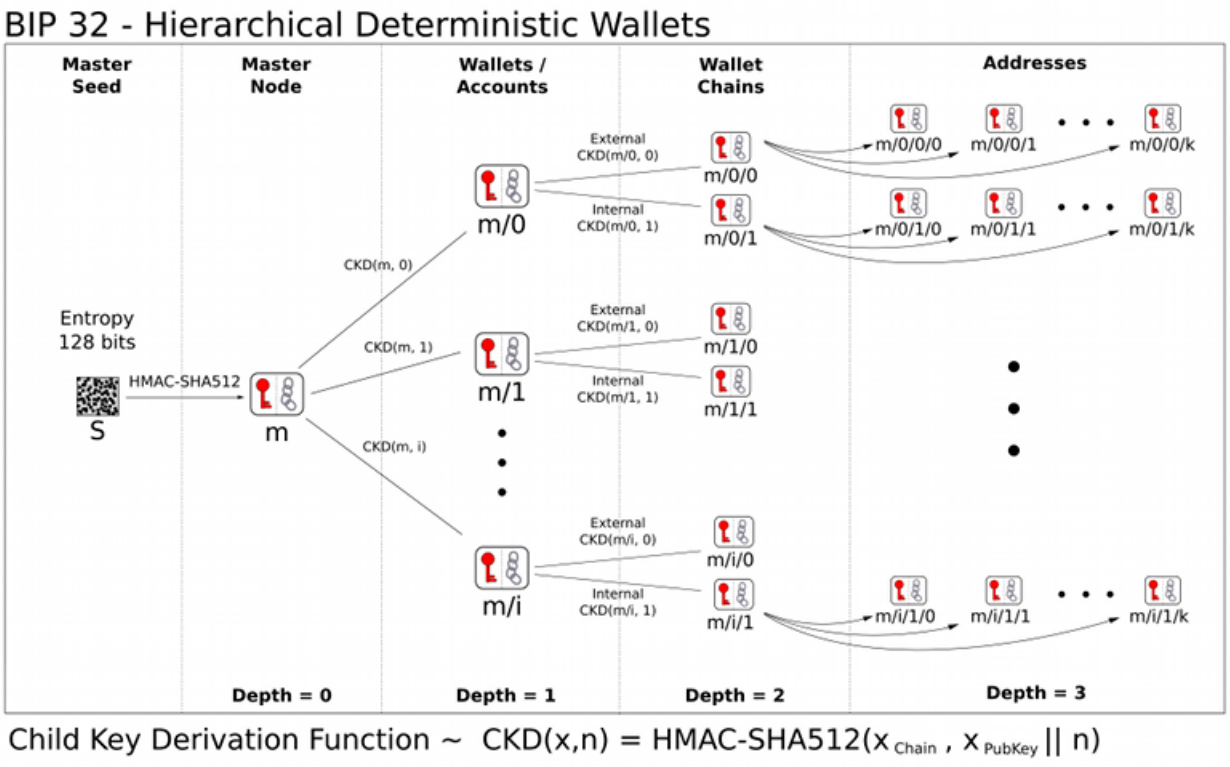

Рис. 8: HD wallet.

Смысл технологии в том, что имея один приватный ключ можно сгенерировать неограниченное количество новых пар приватных и публич-

ных ключей, которые будут однозначно определяться из корневого ключа. При таком подходе ни из какого дочернего ключа невозможно получить родительский, либо другие соседние дочерние ключи, так как они получаются путем применения определенной хэш функции к корневому ключу. Так, для генерации новой пары используется хэш функция HMAC [30], принимающая родительский приватный ключ и некоторое дополнительное уникальное значение. В нашем случае в качестве дополнительного значения будем брать уникальный идентификатор пользователя, зарегистрированного на сервисе. Таким образом, имея корневые приватные ключи для каждого популярного блокчейна, мы однозначно генерируем новый приватный ключ, а следовательно и уникальный адрес (однозначно определяются величины слева направо: приватный ключ, публичный ключ, адрес), который привязывается к конкретному пользователю с определенным «id». Так, при оплате услуги пользователю предлагается свой уникальный адрес для каждой сети. Сканер отследит пополнение адреса средствами, после чего оповестит об этом сервис, который начнет создание, компиляцию, тестирование и развертывание смарт-контракта.

5.5 Основной сценарий работы

Опишем шаги выполнения основного сценария с указанием компонентов, принимающих участие в каждом шаге:

- регистрация и авторизация пользователя (фронтенд AngularJS, Django сервис);
- выбор сети (фронтенд AngularJS);
- ввод данных об адресах (фронтенд AngularJS);
- ввод данных о дате вызова и способе оповещений (фронтенд AngularJS);
- предварительный просмотр данных будущего контракта (фронтенд AngularJS, Django сервис);
- оплата услуги (фронтенд AngularJS, сканер Java, Django сервис, платежный сервис Flask);

- создание и тестирование кода контракта (Django сервис);
- развертывание смарт-контракта (фронтенд AngularJS, Django сервис, подпись транзакций Flask);
- выбор токенов, которые необходимо защитить (фронтенд AngularJS, кошелек MetaMask, Django сервис, сканер Java);
- просмотр готового смарт-контракта, его кода, возможность отмены действия контракта (фронтенд AngularJS, Django сервис).

Глава 6. Анализ полученных результатов

На данный момент состоялся релиз описанного решения, было создано несколько контрактов реальными пользователями как в тестовой, так и в основной сетях. Код всего проекта можно увидеть в репозитории на GitHub [31]. Сравним результаты работы с перечисленными ранее подходами и проектами. Для большей наглядности качественную оценку представим в таблицах (таблица №1 для подходов и таблица №2 для проектов, имеющих схожий интерфейс). Так как реализованный подход уникален, то дальнейшее сравнение полученного решения с существующими сервисами, автоматизирующими процесс развертывания смарт-контрактов, будем осуществлять путем сравнения его с реализованными в них описанными ранее подходами, цель которых идентична.

Таблица 1: Сравнение описанного решения с существующими подходами

	Текущее решение	Хранение средств на разных адресах	Мультисиг контракты	Сид фразы
Хранение средств на контракте	Нет	Нет	Да	Нет
Вероятность потери средств	Низкая	Низкая	Высокая	Средняя
Ограничения аккаунта	Нет	Да	Да	Нет

Таблица 2: Сравнение описанного решения с существующими сервисами

	Текущее решение	Blockcat	Smartz

Интеграция со стандартным кошельком Ethereum	Да	Нет	Нет
Возможность вызова смарт-контрактов, даже если компания перестанет существовать	Да	Нет	Нет
Оплата услуг сервиса большинством известных криптовалют (ETH, BTC, LTC)	Да	Нет	В планах на будущее
Предоставление API для интеграции в сторонних сервисах	Да	Да	Да
Кросс-платформенный графический интерфейс	Да	Плагин для Google Chrome	Да

Так же, для более конкретной оценки результата можно предоставить некоторые количественные оценки сравнения полученного сервиса с существующими в виде гистограмм на Рис. 9.

Таким образом, видно, что решение удовлетворяет поставленным требованиям, а также является быстрым и надежным. Сервис способен конкурировать с существующими реализациями других подходов.

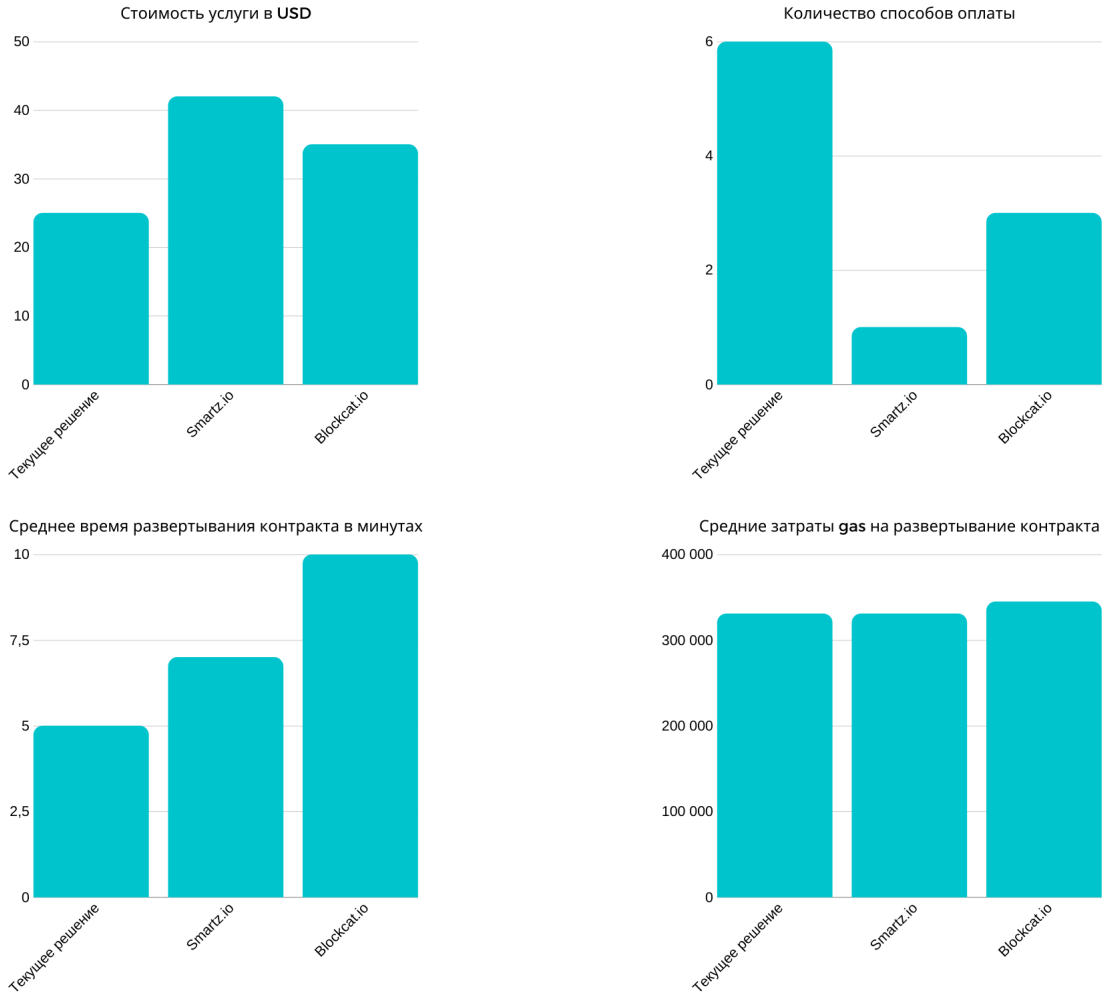

Рис. 9: Сравнение описанного решения с существующими сервисами.

Выводы

Таким образом, блокчейн – очень надежная технология, однако именно из-за этого у пользователей могут возникать определенные трудности с отсутствием возможности восстановления своих приватных ключей. Как выяснилось, такие проблемы можно решать с помощью смарт-контрактов.

Имеются идеи дальнейшего развития проекта. На данный момент таковыми являются рассмотрение возможности вызова смарт-контракта не по достижении определенной даты, а при отсутствии активности у адреса за определенный продолжительный промежуток времени, заранее выбранный пользователем. Сейчас сервис может взаимодействовать только с самым популярным стандартом токенов сети Ethereum – ERC-20, дальней-

шей целью является поддержка остальных стандартов [32]. Также имеет смысл масштабирование и внедрение других блокчейн платформ, например TRON, EOS и NEO.

Заключение

В ходе выполнения выпускной квалификационной работы были достигнуты следующие результаты:

- изучены концепция и структура технологии блокчейн;
- проанализированы существующие решения и выявлены их преимущества и недостатки;
- проведен анализ существующей архитектуры проекта;
- выработана идея решения поставленных проблем;
- выдвинуты технические требования к реализации идеи;
- подобраны инструменты и технологии для выполнения поставленных требований;
- написан, оптимизирован и протестирован код смарт-контракта;
- реализован сервис автоматизации составления и развертывания индивидуальных смарт-контрактов;
- проведен анализ и сравнение полученного решения с существующими продуктами и подходами;
- выявлены пути дальнейшего развития и расширения проекта.

Список литературы

[1] A. Averin, O. Averina Review of Blockchain Technology Vulnerabilities and Blockchain-System Attacks // 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon). 2019.

[2] Официальная статистика сети Ethereum URL: `https://etherscan.io/chart/address` (дата обращения: 19.05.2020).

[3] Badev, An. Baird, M. Brezinski, T. Chen, Cl. Ellithorpe, M. Fahy, L. Kargenian, V. Liao, K. Malone, B. Marquardt, J. Mills, D. Ng, W. Ravi, An. Wang, K. (2016). Distributed Ledger Technology in Payments, Clearing, and Settlement. Finance and Economics Discussion Series. 2016. doi:10.17016/FEDS.2016.095.

[4] Vinayak Singla, Indra Kumar Malav, Jaspreet Kaur, Sumit Kalra Develop Leave Application using Blockchain Smart Contract // 2019 11th International Conference on Communication Systems Networks (COMSNETS). 2019.

[5] Zil, K., Strazdin a, R. (2018). Blockchain Use Cases and Their Feasibility. Applied Computer Systems, 23(1), 12–20. doi:10.2478/acss-2018-0002

[6] Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H. (2017). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. 2017 IEEE International Congress on Big Data (BigData Congress). doi:10.1109/bigdatacongress.2017.85

[7] Официальная документация Solidity URL: `https://solidity.readthedocs.io/en/v0.6.7/` (дата обращения: 17.04.2020).

[8] Официальная документация Uml2Solidity URL: `https://github.com/UrsZeidler/uml2solidity` (дата обращения: 17.04.2020).

[9] Официальный сайт сервиса Etherscripter URL: `https://etherscripter.com/0-5-1/` (дата обращения: 17.04.2020).

[10] Библиотека смарт-контрактов OpenZeppelin URL: https://openzeppelin.com/ (дата обращения: 17.04.2020).

[11] Официальный сайт сервиса Unibright.io URL: https://unibright.io/ (дата обращения: 17.04.2020).

[12] Официальный сайт сервиса Smartz.io URL: https://smartz.io/ (дата обращения: 17.04.2020).

[13] Официальный сайт сервиса Blockcat.io URL: https://blockcat.io/ (дата обращения: 17.04.2020).

[14] Luu, L., Chu, Duc-Hiep, Olickel, H., Saxena, Pr., Hobor, Aq. (2016). Making Smart Contracts Smarter. 254-269. doi:10.1145/2976749.2978309.

[15] Официальный сайт сервиса MyWish.io URL: https://mywish.io/ (дата обращения: 17.04.2020).

[16] Официальный сайт сервиса SWAPS.NETWORK URL: https://swaps.network/ (дата обращения: 14.04.2020).

[17] Официальный сайт кошелька MetaMask URL: https://metamask.io/ (дата обращения: 14.04.2020).

[18] Официальный сайт кошелька MyEtherWallet URL: https://www.myetherwallet.com/ (дата обращения: 11.04.2020).

[19] Bartoletti, M., Pompianu, L. (2017). An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. Lecture Notes in Computer Science. doi:10.1007/978-3-319-70278-0 31.

[20] Gavin Wood. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper.

[21] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine // 2018 IEEE 31st Computer Security Foundations Symposium (CSF). 2018.

[22] Ethereum Yellow Paper URL: http://paper.gavwood.com/ (дата обращения: 25.04.2020).

[23] Среда для разработки и отладки смарт-контрактов в локальной сети Ethereum URL: https://remix.ethereum.org/ (дата обращения: 14.05.2020).

[24] Криптовалютная биржа Binance // Криптовалютная биржа Binance URL: https://www.binance.com/ru (дата обращения: 18.05.2020).

[25] Zhenglin Liu, Xin Dong, Yizhi Zhao, Dongfang Li Hardware implementation of SHA-3 candidate based on BLAKE-32 // 2012 5th International Conference on BioMedical Engineering and Informatics. 2012.

[26] Token Saver Smart Contract // GitHub URL: https://github.com/MyWishPlatform/token_saver (дата обращения: 20.05.2020).

[27] Ethereum White Paper URL: https://github.com/ethereum/wiki/wiki/White-Paper (дата обращения: 20.04.2020).

[28] Открытый реестр транзакций сети Ethereum "Etherscan"URL: https://etherscan.io/ (дата обращения: 15.05.2020).

[29] Dmitry Khovratovich, Jason Law BIP32-Ed25519: Hierarchical Deterministic Keys over a Non-linear Keyspace // 2017 IEEE European Symposium on Security and Privacy Workshops (EuroSPW). 2017.

[30] Selman Yakut, A. Bedri Özer HMAC based one tıme password generator // 2014 22nd Signal Processing and Communications Applications Conference (SIU). 2014.

[31] Token Saver Project // GitHub URL: https://github.com/MyWishPlatform/mywill_backend (дата обращения: 20.05.2020).

[32] Ethereum Improvement Proposals URL: https://eips.ethereum.org/erc (дата обращения: 18.04.2020).