МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

_____ Л.Б. Соколинский

«___»_____ 2024 г.

Разработка расширения протокола XMPP для согласования ключей шифрования с аутентификацией

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 09.03.04.2024.306-05-097.ВКР

Научный руководитель,
доцент кафедры СП, к.п.н.
_____ О.Н. Иванова

Автор работы,
студент группы КЭ-433
_____ Е.М. Короткова

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
29.01.2024 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студентке группы КЭ-433
Коротковой Екатерине Михайловне,
обучающейся по направлению
09.03.04 «Программная инженерия»

1. **Тема работы** (утверждена приказом ректора от 22.04.2024 г. № 764-13/12)
Разработка расширения протокола XMPP для согласования ключей шифрования с аутентификацией.
2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**
3.1. Saint-Andre P. RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core. // Internet Engineering Task Force (IETF), 2011. – 210 с.
3.2. Saint-Andre P. RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. // Internet Engineering Task Force (IETF), 2011. – 113 с.
3.3. Svetlin Nakov. Practical Cryptography for Developers. // SoftUni (Software University), 2018. [Электронный ресурс] URL: https://cryptobook.nakov.com (дата обращения: 10.02.2024 г.).
4. **Перечень подлежащих разработке вопросов**
4.1. Провести анализ предметной области и обзор существующих протоколов XMPP.
4.2. Разработать алгоритм аутентификации пользователей и верификации их устройств с минимальным взаимодействием пользователя с клиентом.
4.3. Написать текст документации протокола XEP.

4.4. Разработать прототип реализации протокола согласования ключей шифрования с аутентификацией пользователей путем передачи XML-станз.
4.5. Реализовать протокол XEP на iOS-клиенте приложения.
4.6. Протестировать работу протокола с другими клиентами приложения.
5. Дата выдачи задания: 29.01.2024 г.

Научный руководитель,
доцент кафедры СП, к.п.н. О.Н. Иванова

Задание принял к исполнению Е.М. Короткова

ГЛОССАРИЙ

1. *Идентификатор* [1] – это уникальный признак, позволяющий выделить и опознать какой-либо объект среди множества других подобных объектов.

2. *Устройство* [2] – конкретный экземпляр клиента.

3. *Аутентификация* [3] – это процесс проверки и подтверждения личности пользователя в системе, который обеспечивает безопасность и конфиденциальность данных.

4. *Верификация* [4] – это процесс проверки и подтверждения подлинности, достоверности или правильности чего-либо.

5. Шифрование [5] – это метод преобразования данных, пригодных для чтения человеком, в форму, которую человек не сможет прочитать. За счет этого данные остаются конфиденциальными и приватными.

6. Дешифрование [5] – это метод преобразования данных, не пригодных для чтения, в форму, которую человек сможет прочитать.

7. *Публичный ключ* [5] – это общедоступный индивидуальный ключ устройства, используемый для шифрования данных или верификации подписи данных.

8. *Приватный ключ* [5] – это секретный индивидуальный ключ устройства, используемый для расшифровки данных или подписи данных.

9. *Общий ключ* [5] – ключ, используемый для симметричного шифрования и дешифрования, известный всем участникам процесса шифрования.

10. *Вектор инициализации* [6] – произвольная последовательность байтов, которая используется вместе с секретным ключом для шифрования данных алгоритмом AES-CBC.

11. *Доверие* [7] – отношение между устройствами, которые доказали подлинность своих ключей между собой.

12. *Компрометация данных* [8] – это процесс несанкционированного доступа к информации или утечки конфиденциальных данных.

13. *Хэш-функции* [9] – это криптографические инструменты, которые играют фундаментальную роль в обеспечении целостности и безопасности цифровой информации. Эти функции принимают входные данные (или «сообщение») и создают строку символов фиксированного размера, обычно в шестнадцатеричном или двоичном представлении.

14. *Энтропия* [10] – мера, которая измеряет степень неопределенности информации и показывает, насколько сложно восстановить открытый текст из зашифрованного сообщения без знания ключа.

15. *JID (Jabber IDentifier)* [11] – идентификатор, который однозначно идентифицируют пользователей в сети XMPP.

16. *XML Станза* [12] – небольшие фрагменты структурированных данных для обмена между двумя (или более) сущностями.

17. *IQ (Info / Query)* [12] – тип станзы, имеющий «request-response» механизм. IQ позволяет сущности сделать запрос и получить ответ от другой сущности, к которой был адресован IQ.

18. *Message* [12] – тип станзы, имеющий «push» механизм, посредством которого одна сущность отправляет информацию другой сущности.

19. *Presence* [12] – тип станзы, имеющий «publish-subscribe» механизм, посредством которого несколько пользователей получают информацию о доступности сети пользователя, на которого они подписались.

20. *XEP (XMPP Extension Protocols)* [13] – расширения протокола XMPP, добавляющие новые функции и возможности.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность

В современном мире, где информационные технологии играют решающую роль во многих сферах жизни, безопасность данных становится все более важной. Шифрование – один из основных элементов безопасности данных, обеспечивающий конфиденциальность и целостность информации.

Протокол XMPP широко используется для обмена сообщениями в реальном времени. Однако, в чистом виде протокол передает все данные в виде, при котором переписка доступна операторам серверов. По этой причине, при использовании протокола применяют протоколы конечного шифрования – end-to-end encryption (e2ee) [14]. Чтобы проверить, является ли устройство, с которым пользователь хочет установить безопасное соединение с шифрованием e2ee, подлинным, необходимо проверять его цифровой отпечаток вручную, что является сложным и неудобным процессом. Поэтому пользователи часто пренебрегают требованиями безопасности и устанавливают шифрованное соединение вслепую, не сверив цифровые отпечатки и не убедившись в подлинности конечного устройства.

Разрабатываемый протокол для согласования ключей шифрования с аутентификацией, делает этот процесс выяснения подлинности устройств и аутентификации пользователей более автоматизированным. Все действия пользователей сводятся к сообщению определенного кода одним пользователем другому, который должен ввести этот код на своем устройстве. Это сделает процедуру аутентификации пользователей и верификацию устройств менее трудоемким процессом.

Постановка задачи

Целью выпускной квалификационной работы является разработка расширения протокола XMPP для согласования ключей шифрования с аутентификацией.

Для достижения поставленной цели необходимо будет решить следующие задачи.

1. Провести анализ предметной области и обзор существующих протоколов XMPP.
2. Разработать алгоритм аутентификации пользователей и верификации их устройств с минимальным взаимодействием пользователя с клиентом.
3. Написать текст документации протокола XEP.
4. Разработать прототип реализации протокола согласования ключей шифрования с аутентификацией пользователей путем передачи XML-станз.
5. Реализовать протокол XEP на iOS-клиенте приложения.
6. Протестировать работу протокола с другими клиентами приложения.

Структура и содержание работы

Работа состоит из введения, трех глав, заключения и списка литературы. Объем работы составляет 59 страниц, объем списка литературы – 24 источника.

В первой главе описывается предметная область, а также проводится анализ существующих аналогов и методов, решающих поставленные задачи.

Вторая глава посвящена описанию функциональных и нефункциональных требований к разрабатываемому приложению, и проектированию основного алгоритма протокола.

В третьей главе описана реализация разработанного протокола и хранилища, протокола обмена доверенными устройствами.

В приложении А содержится код основных методов менеджеров двух реализованных протоколов и код реализованного прототипа.

В приложении Б содержатся скриншоты реализованной функциональности приложения, т. е. верификации устройств с обеих сторон (отправляющей и принимающей), и диаграммы последовательностей протоколов.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание предметной области

Криптография предоставляет средства цифровой подписи сообщений, которые гарантируют подлинность и целостность сообщения [15]. Для создания и верификации цифровой подписи используется ключевая пара. Она представляет из себя два набора байт, сформированных средством электронной подписи.

Первый из них – это ключ электронной подписи, который называют «приватным». Он используется для создания самой цифровой подписи и должен быть известен только пользователю, обладающему ключевой парой.

Второй набор байтов – это ключ верификации подписи, который называют «публичным». Каждый «публичный» ключ привязан к своему «приватному» ключу и необходим для проверки цифровой подписи и целостности подписанных данных.

В настоящее время, в области передачи данных наибольшее распространение получило асимметричное шифрование. В отличии от симметричного шифрования, при асимметричном не нужно обмениваться общим ключом, защищая его при этом и тратя дополнительное количество циклов обмена данными.

Хэш-функции – это важные инструменты в области криптографии, которые обеспечивают целостность и безопасность цифровой информации. Они принимают входные данные и генерируют строку символов фиксированного размера, обычно в шестнадцатеричном или двоичном формате. Это позволяет эффективно проверять целостность данных и создавать уникальные идентификаторы для файлов и сообщений. Примером хэш-функции может служить SHA-256, который используется для хеширования паролей и подписей в блокчейне.

В приведенном ниже примере (рисунок 1) строка «John Smith» хэшируется с хэш-значением 02, а «Liza Smith» хэшируется со значением 01.

Входные строки «John Smith» и «Sandra Dee» хэшируются со значением 02, это называется «коллизией».

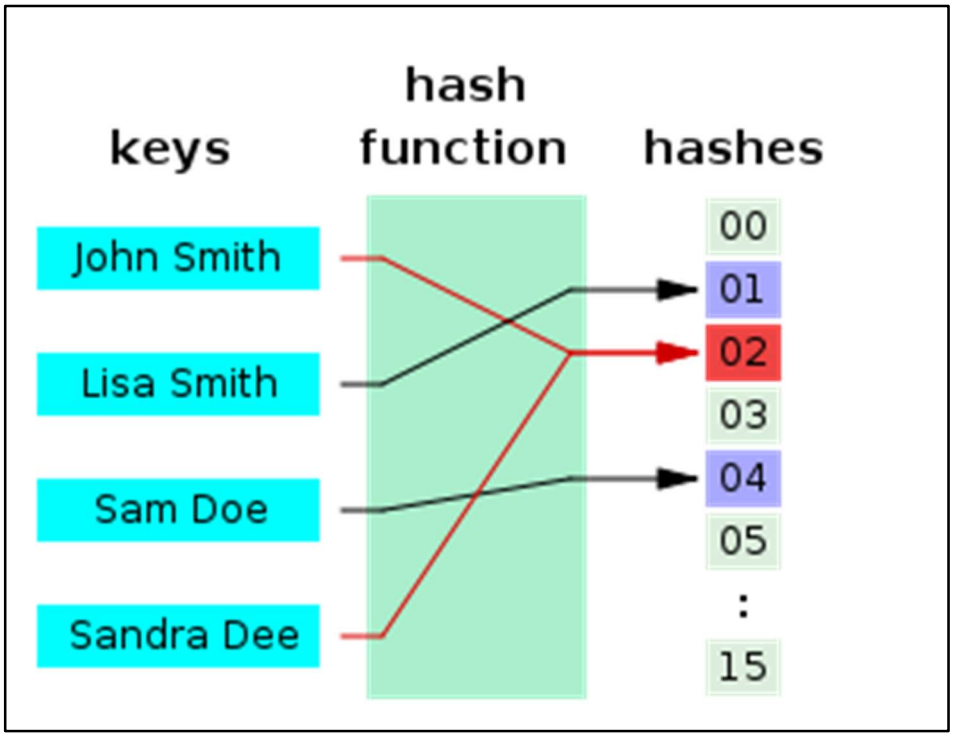

Рисунок 1 – Пример коллизии в хэшировании

Криптографические хэш-функции преобразуют текстовые или двоичные данные в хеш-значение фиксированной длины и устойчивы к коллизиям и необратимы, поэтому они будут использоваться в протоколе, разработка которого является целью данной работы.

XMPP (eXtensible Messaging and Presence Protocol) – открытый протокол, основанный на XML (eXtensible Markup Language), который используется для мгновенного обмена сообщениями и информацией о присутствии в режиме, близком к реальному времени. Помимо передачи текстовых сообщений, XMPP поддерживает передачу аудио, видео и других файлов по сети. Протокол XMPP принят IETF как RFC 6120 [12] и RFC 6121 [16].

XMPP работает на клиент-серверной архитектуре (рисунок 2). Это означает, что, когда пользователь отправляет сообщение через XMPP протокол получателю, оно сначала отправляется на сервер отправителя, затем

сервер направляет его нужному получателю (или серверу получателя, от сервера сообщение далее уходит на клиент получателя).

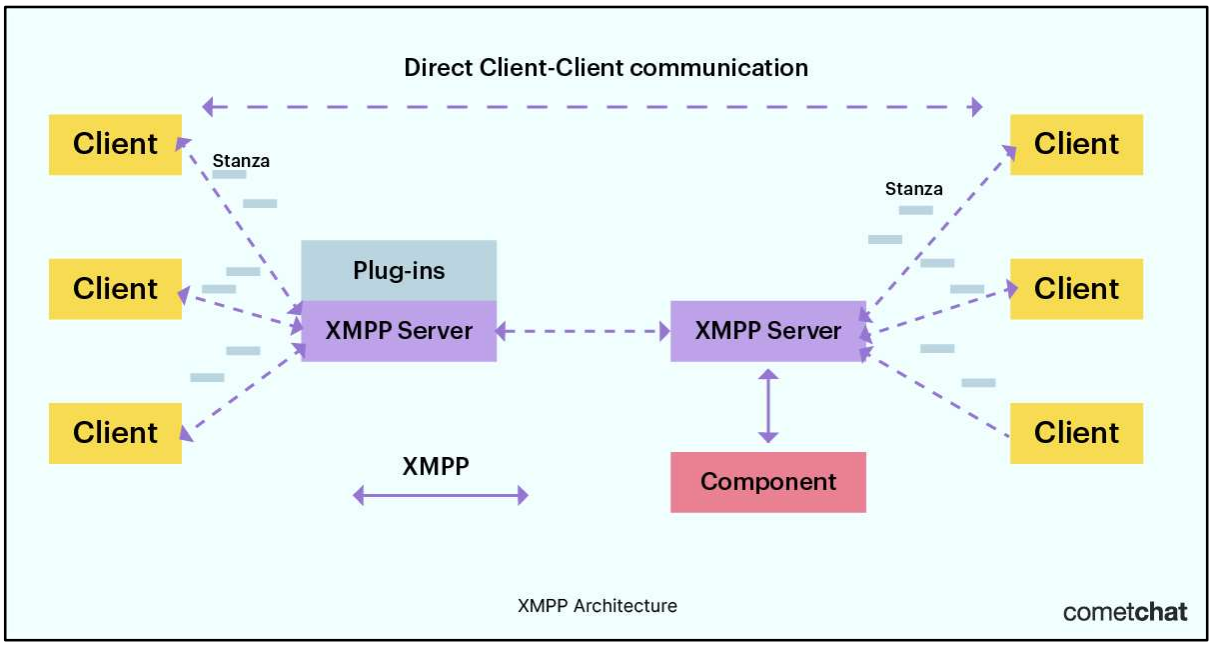

Рисунок 2 – Клиент-серверная архитектура протокола XMPP

XEP (Xmpp Extensions Protocol) [13] – расширение протокола XMPP. Каждый XEP описывает конкретную задачу, такую как форматирование сообщений, передача файлов, многопользовательские чаты и многое другое. К примеру, расширениями достигается передача видео, аудио и других файлов. Они предоставляют стандартный формат для всех, кто хочет использовать определенную функциональность в рамках протокола XMPP.

Одним из таких расширений являются протоколы конечного шифрования, которые будут использоваться в разрабатываемом протоколе. Шифрование – это процесс обратимого преобразования данных в целях их сокрытия от посторонних лиц, но в то же время предоставления доступа авторизированным пользователям. Шифрование данных, передающихся по какому-либо каналу, обеспечивает их конфиденциальность.

Схемы шифрования бывают [5]:

1) симметричные – используют один и тот же ключ для шифрования и дешифрования сообщений;

2) ассиметричные – используют пару ключей: открытый ключ (ключ шифрования) и соответствующий закрытый ключ (ключ дешифрования).

В XMPP существует три основных схемы сквозного шифрования: обмен сообщениями Off-the-Record (XEP-0364: Current Off-the-Record Messaging Usage [17]), OpenPGP (XEP-0027: Current Jabber OpenPGP Usage [18]) и OMEMO Encryption [2]. Старые версии OTR имели существенные недостатки в плане удобства использования. Например, история чата не синхронизировалась между другими устройствами участвующих сторон. Более того, чаты OTR были возможны только в том случае, если оба участника были онлайн одновременно, из-за того, как работала схема соглашения о ключах OTR. OpenPGP не обеспечивает никакой прямой секретности и уязвим для атак повторного воспроизведения.

Протокол OMEMO Encryption использует схему шифрования Double Ratchet для обеспечения шифрования между несколькими устройствами, позволяя безопасно синхронизировать сообщения между ними, даже если некоторые из них находятся в оффлайне.

В настоящей работе описывается разработка XEP, который будет модифицировать существующий протокол шифрования XEP-0384: OMEMO Encryption. Благодаря разрабатываемому протоколу, процедура шифрования станет более безопасной для пользователей.

1.2. Сравнительный анализ аналогов

TLS протокол

TLS (Transport Layer Security) [19] – это протокол, который защищает данные во время их передачи по Сети. Для обеспечения защиты данных, TLS создает специальный канал для передачи данных, где они не могут быть прочитаны или изменены без использования секретного ключа.

Кроме шифрования, в TLS используются различные алгоритмы хеширования. Когда ваш браузер отправляет данные через протокол TLS, вместе с данными автоматически отправляется их хеш. Сервер получает данные и пропускает их через ту же функцию хеширования, что и браузер. Если хеши совпадают, значит данные не были изменены.

TLS обеспечивает проверку подлинности сервера с помощью специальных сертификатов, которые подтверждает уполномоченный центр сертификации.

TLS представляет собой не один, а два протокола: протокол рукопожатия и протокол записи. Каждый из них выполняет свой набор функций. Протокол рукопожатия состоит из следующих этапов:

1) представление и договор об алгоритмах шифрования;
2) обмен ключами;
3) завершение рукопожатия.

После завершения рукопожатия применяется протокол записи. Он отвечает за шифрование и передачу данных между сайтом и браузером. Он работает на основе установленного соединения согласно параметрам, о которых клиент и сервер договорились во время рукопожатия. Этот процесс происходит в несколько этапов, в ходе которых данные обрабатываются следующими операциями:

1) фрагментация;
2) компрессия;
3) шифрование;
4) подпись;
5) передача.

Этот протокол не подходит под цель данной работы из-за того, что в нем идентификация узла осуществляется за счет сертификатов, которые должны получать все узлы сети интернет у доверенных центров сертификации. Протокол XMPP не предусматривает получение пользователями сер-

тификатов безопасности, и не существует общепризнанной сети удостоверяющих центров, которые бы могли выполнять роль верификации пользователей. На данный момент используется ручная верификация устройств путем сравнения их цифровых отпечатков.

Сравнение цифровых отпечатков устройств

В XMPP при шифровании OMEMO верификация устройств пользователей происходит путем сравнения их цифровых отпечатков между собой. У каждого такого устройства имеется уникальный цифровой отпечаток, который открыт для всех. У пользователя на устройстве отображается как свой цифровой отпечаток, так и цифровой отпечаток устройства собеседника, с которым он хочет обмениваться шифрованными сообщениями.

Если пользователь хочет проверить, является ли устройство, с которым он хочет установить шифрование, устройством собеседника, а не злоумышленника, он должен сравнить отображаемые отпечатки. Если цифровые отпечатки совпадают, пользователи могут быть уверены, что устройства, между которыми они хотят установить шифрование, являются подлинными.

Недостатком этого процесса является то, что он долгий и трудоемкий, так как каждый цифровой отпечаток устройства имеет длину 64 символа. Кроме того, что поочередное сравнение четырех цифровых отпечатков занимает много времени, в процессе могут быть совершены ошибки, из-за которых пользователь может установить шифрование не с тем устройством. Кроме такого способа проверки подлинности устройства есть еще и другие, например, проверка общим вопросом в протоколе OTR, но этот способ не является достаточно безопасным.

Вывод по первой главе

В этой главе были рассмотрены предметная область и аналоги разрабатываемого протокола, что позволило определить, какие функции должен выполнять разрабатываемый протокол.

2. ПРОЕКТИРОВАНИЕ

2.1. Определение требований

Функциональные требования

В ходе проектирования приложения были определены следующие ниже функциональные требования, определяющие действия разрабатываемой системы.

1. Алгоритм должен быть устойчив к атаками типа MIDM (Man In The Middle).

2. Секретные данные, которые не должны быть известны никому, кроме конечных пользователей, участвующих в процессе аутентификации, должны передаваться в зашифрованном виде.

3. Данные, которые можно перехватить и/или изменить, должны передаваться подписанными.

4. Процесс аутентификации должен проходить автоматически, с минимальным включением пользователя. Пользователь только сообщает секретный код другому пользователю или вводит сообщенный другим пользователем секретный код на своем устройстве.

5. Отсутствующие в сети устройства в процессе аутентификации пользователей и верификации их устройств при появлении в сети должны получить информацию о новых доверенных ключах.

Нефункциональные требования

В ходе проектирования приложения были определены следующие ниже нефункциональные требования, от которых зависят средства и способы реализации.

1. Протокол должен корректно работать между различными клиентами приложения.

2. При неудачной аутентификации пользователей на стороне одного устройства, устройству на другой стороне должна посылаться информация о неудачном окончании сессии аутентификации.

3. На всех устройствах, вовлеченных в процесс верификации, должны быть включены Message Carbons [20].

4. При подтверждении запроса на аутентификацию пользователей и верификацию устройств одним устройством, остальные устройства, получившие запрос, не должны иметь возможности ответить на этот запрос.

2.2. Алгоритм верификации устройств

Протокол Диффи-Хеллмана

Протокол Диффи-Хеллмана [21] является протокол обмена ключами, который позволяет двум сторонам безопасно согласовать общий секретный ключ в открытом канале связи (рисунок 3).

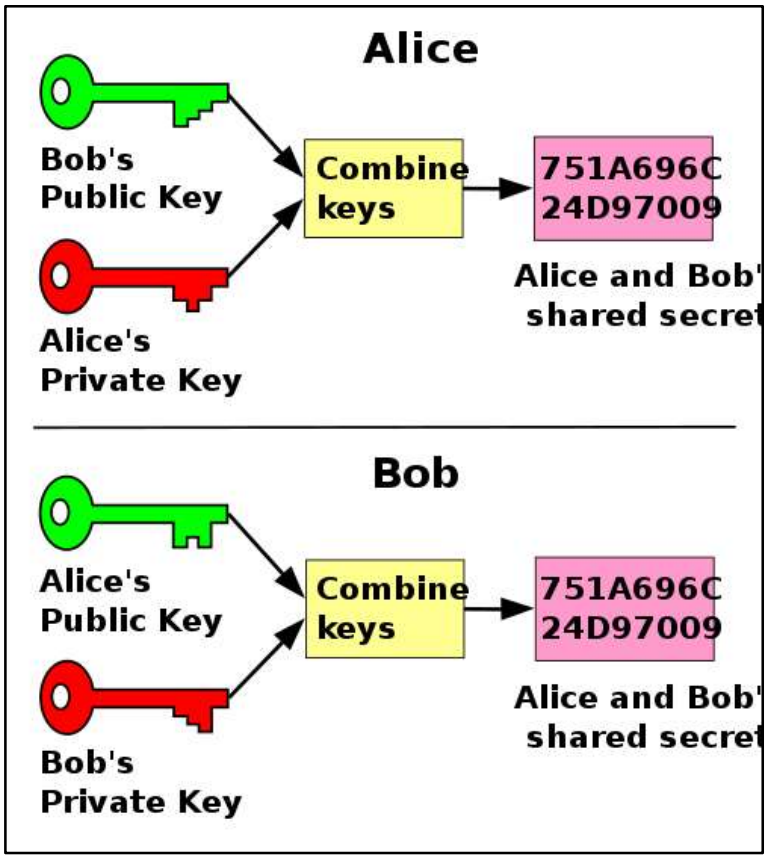

Рисунок 3 – Схема работы протокола DH

Диффи-Хеллман основан на принципе неполного обмена ключом шифрования через сеть. Каждая сторона обладает публичным ключом (доступным для всех, включая злоумышленника) и приватным ключом (который знает только владелец этого ключа).

Недостатком этого протокола является то, что в канал связи может вмешаться злоумышленник, назовем его Мэтью, установив связь с Алисой, прикинувшись Бобом, и установив связь с Бобом, притворившись Алисой (рисунок 4). У Мэтью с Алисой будет один общий ключ шифрования, которым он сможет расшифровать сообщение, прочитать, и зашифровать общим с Бобом ключом шифрования. Так, Мэтью сможет узнать, что находится в секретном сообщении, в то время как Алиса и Боб не будут об этом догадываться.

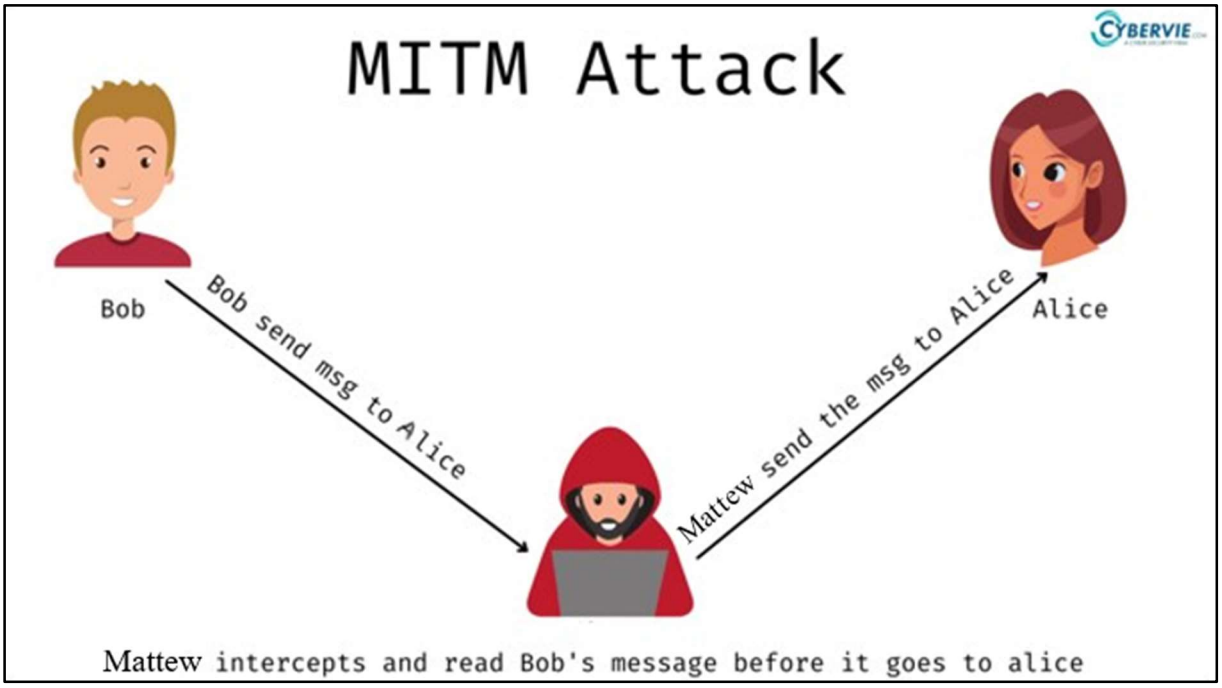

Рисунок 4 – Атака MITM

Таким образом, протокол Диффи-Хеллмана подходит для установления безопасного соединения, но не решает проблему аутентификации пользователей, что является существенным недостатком для безопасного обмена ключами между устройствами в сети XMPP.

Протокол аутентификации пользователей и верификации устройств

В результате обзора возможных методов атак злоумышленников, был спроектирован алгоритм верификации двух устройств. В результате работы алгоритма два устройства установят доверительную связь.

Алгоритм был спроектирован на базе протокола Диффи-Хеллмана. В алгоритм включена еще и двусторонняя аутентификация. Аутентификация заключается в том, что оба пользователя будут знать секретный код, который будет передаваться не по каналу связи, в котором проходит верификация устройств, а в каком-либо другом канале связи, в идеале лично или по звонку. Так, злоумышленник, вмешавшийся в канал связи, не сможет узнать этот код.

Пусть два пользователя, Алиса и Боб, хотят верифицировать свои устройства A и B между друг другом. У каждого устройства имеется id, доверительный ключ (вычисляется путем конкатенации id устройства и его цифрового отпечатка – формула 1), публичный ключ и приватный ключ.

Два устройства могут создать общий для них ключ, который используется в дальнейшем для зашифровки и расшифровки сообщений. Этот ключ вычисляется из приватного ключа пользователя и публичного ключа собеседника (формулы 2 и 3). У двух устройств этот ключ будет одинаковым (формула 4).

$$trustedKey = deviceId + deviceFingerprint, \quad (1)$$

$$sharedKeyAlice = getSharedKey(privateKeyAlice, publicKeyBob), \quad (2)$$

$$sharedKeyBob = getSharedKey(privateKeyBob, publicKeyAlice), \quad (3)$$

$$sharedKeyAlice = sharedKeyBob. \quad (4)$$

Для обеспечения аутентификации пользователей будет вычислен другой ключ шифрования от общего ключа шифрования и от секретного кода, известного только двум пользователям (формула 5):

$$encryptionKey = hash256\big(sharedKey + hash256(code)\big). \quad (5)$$

На рисунке 1 приложения Б представлена диаграмма последовательности процесса верификации двух устройств разных пользователей.

Верификация начинается с того, что пользователь Алиса с устройства A отправляет Бобу запрос на верификацию. Запрос приходит на все его

устройства, принять он может этот запрос с любого устройства. Сессия верификации начнется с тем устройством Боба, с которого он принял запрос на верификацию.

После согласия на верификацию устройство B генерирует секретный шестизначный цифробуквенный код, который Боб должен будет передать Алисе вне установленного канала связи между устройствами A и B, и случайную байтовую последовательность длиной 16 байт. От кода, соединенным со случайной байтовой последовательностью, будет рассчитываться хэш (это нужно, чтобы понять, что другой пользователь знает секретный код, не раскрывая его при этом). Случайная байтовая последовательность вычисляется для обоих устройств, нужна для увеличения энтропии хэша и должна быть известна только двум устройствам (поэтому передается зашифрованной). Устройство B зашифровывает случайную байтовую последовательность (далее b) ключом $encryptionKey$ (формула 4) и передает ее устройству A.

Устройство A расшифровывает b ключом $encryptionKey$. Далее генерирует свою случайную байтовую последовательность (далее a) и зашифровывает ее. Теперь устройство A должно доказать устройству B, что оно знает секретный код, при этом не раскрывая его. Для этого устройство рассчитывает хэш SHA256 от своего доверительного ключа (формула 1), секретного кода и b, после чего зашифровывает этот хэш. Зашифрованные a и хэш передаются устройству B.

Устройство B расшифровывает полученные b и хэш. Рассчитывает хэш от доверительного ключа устройства A, секретного кода и b на своей стороне. Рассчитанный хэш сравнивается с полученным, чтобы убедиться в том, что устройство A знает секретный код. Если хэши оказываются равными, то устройство B убеждается в том, что устройство A является подлинным устройством Алисы. Теперь устройству B надо доказать устрой-

ству A, что оно является подлинным устройством Боба. Для этого оно рассчитывает другой хэш от своего доверительного ключа, секретного кода, b и a, зашифровывает этот хэш и передает устройству A.

Устройство A расшифровывает полученный хэш, рассчитывает хэш от доверительного ключа устройства B, секретного кода, b и a, после чего сравнивает этот рассчитанный хэш с полученным. Если они оказываются равными, то устройство A тоже убедилось в том, что устройство B является подлинным устройством Боба.

Теперь оба устройства могут запросить информацию обо всех доверенных устройствах друг друга: устройство A обо всех доверенных устройствах B, принадлежащих Бобу ($B2$, $B3$, $B4$, …), а устройство В обо всех доверенных устройствах A, принадлежащих Алисе ($A2$, $A3$, $A4$, …). Для этого процесса был использован другой протокол, описывающий правила обмена информацией о доверенных устройствах.

Процесс верификации между двумя устройствами одного пользователя происходит таким же образом, отличается лишь обмен информацией о доверенных устройствах: два устройства одного пользователя обмениваются доверенными устройствами других контактов.

Протокол обмена информацией о доверенных устройствах

Обмен информацией о доверенных устройствах может быть двух типов:

– распространение информации об устройствах других контактов, которым доверяет устройство;

– распространение информации о других устройствах пользователя, которым доверяет устройство.

Устройство может обмениваться информацией об устройствах других контактов, которым доверяет, только с устройствами своего пользователя (рисунок 5 приложения Б). Этот список не доступен другим контактам. Эта информация подписывается и помещается в контейнер сообщения вместе с

подписью. При получении этого сообщения устройства этого же пользователя проверяют подпись сообщения, и в случае, если подпись верифицирована успешно, добавляют каждое устройство из полученного сообщения в список своих доверенных устройств. После этого, для каждого из полученных устройств запрашивается публикация с другими устройствами владельца этого устройства, которым оно доверяет. Устройства из полученного списка также добавляются в список доверенных устройств.

При распространении устройством информации о своих доверенных устройствах, которые принадлежат этому же владельцу устройства, оно публикует (XEP-0060: Publish-Subscribe [22]) на сервер список с этими устройствами (рисунок 6 приложения Б). Информация о том, что произошла публикация, приходит всем устройствам, которые подписаны на владельца устройства. Вместе со списком устройств публикуется подпись, которой подписывается этот список. При получении информации о новой публикации, устройства проверяют подпись. В случае успешной верификации подписи, устройства добавляют устройства из публикации в список своих доверенных устройств.

Вывод по второй главе

В этой главе были выделены функциональные и нефункциональные требования к проектируемой системе, был описан алгоритм верификации двух устройств разных пользователей (по такому же принципу работает алгоритм верификации двух устройств одного пользователя), алгоритм обмена списками доверенных устройств для избежания лишних верификаций устройств, которые уже верифицированы доверенным устройством.

3. РЕАЛИЗАЦИЯ И ТЕСТИРОВАНИЕ СИСТЕМЫ

Для реализации протокола на iOS-клиенте использовался язык Swift. Написание программного кода и его отладка осуществлялась в среде разработки Xcode 15.2.

Для реализации протокола XMPP в работе будет использоваться XMPPFramework [23]. Этот фреймворк предоставляет базовую реализацию стандарта XMPP, а также инструменты, необходимые для чтения и записи XML. XMPPFramework содержит самые популярные и используемые протоколы.

Для генерации ключевой пары и создания общего секретного ключа шифрования была выбрана кривая Curve25519 [24]. Для подписи данных будет использоваться алгоритм цифровой подписи на эллиптических кривых (ECDSA) Этот алгоритм позволяет подписывать данные, создавая подпись, которую можно верифицировать для подтверждения подлинности данных и уверенности в том, что их не изменяли.

Шифрование данных будет использоваться с использованием алгоритма AES в режиме CBC [6]. Для шифрования и дешифрования данных будет необходимо использовать кроме публичного ключа и самих данных, еще и вектор инициализации. Он будет передаваться вместе с зашифрованными данными, чтобы с помощью этого вектора их можно было дешифровать.

3.1. Реализация прототипа протокола

Для симуляции работы нескольких клиентов одновременно был разработан прототип на языке Python. Код программы содержит несколько классов: `Bot` для представления аккаунта пользователя клиента, `Opponent` для представления собеседника. В корневой функции проекта `main()` (листинг 1) инициализируется класс Bot, представляющий пользователя устройства.

Листинг 1 – Код метода `main()`

```
async def main():
    user_jid, user_password, opponent_jid = choose_user()
    alice = bot.Bot(
        user_jid,
        user_password
    )
    alice.is_key_press_needed = True
    await alice_connected(alice)
```

Из функции вызывается метод `alice_connected()` (листинг 2), который публикует бандл с ключами, обновленный список устройств с текущим, запрашивает с сервера список устройств, поддерживающих OMEMO шифрование. Далее метод ожидает действия пользователя. В зависимости от действия пользователя, клиент отправит запрос на верификацию любому устройству контакта из списка доступных.

Листинг 2 – Код метода `alice_connected()`

```
async def alice_connected(alice: bot.Bot):
    await initialize_user_devices(alice)
    await alice.publish_bundle()
    await alice.publish_devices()
    await alice.get_omemo_devices(alice.jid)
    print(f"JID: {alice.client.local_jid}")
    await alice.print_home_screen()
    while True:
        if alice.is_key_press_needed:
            alice.is_key_press_needed = False
            if sys.platform == 'win32':
                alice.task = asyncio.get_event_loop().create_task(al-
ice.wait_user_key_press())
            else:
                alice.task = asyncio.create_task(al-
ice.wait_user_key_press())
            await alice.task
        await asyncio.sleep(1)
```

При выборе устройства для верификации, вызывается метод `get_message_verify_request()` (листинг 3), который инициализирует экземпляр сообщения для отправления устройству контакта.

Листинг 3 – Код метода `get_message_verify_request()`

```
def get_message_verify_request(self, to_jid) -> aioxmpp.Message:
        message = aioxmpp.Message(
            to=to_jid,
            from_=self.jid,
            type_=aioxmpp.MessageType.CHAT,
        )
        message.xep0000 = xso.AuthenticatedKeyExchange()
```

```
        message.xep0000.sid = str(uuid.uuid4())
        message.xep0000.verification_start = xso.VerificationStart(self.de-
vice_id)
        return message
```

Для обработки сообщений с информацией о сессии верификации написан обработчик сообщений `message_received()` (листинг 6 приложения А). В нем обрабатываются все входящие сообщения о верификации, и, в зависимости от этапа сессии верификации, выполняется определенный шаг алгоритма и высылается необходимая на этом этапе станза.

3.2. Реализация основного класса протокола

Для управления протоколом был создан класс `AuthenticatedKey-ExchangeManager`, который наследуется от класса `AbstractXMPPManager`, от которого наследуются все классы, реализующие разные протоколы XMPP на этом клиенте.

Класс `AbstractXMPPManager` содержит два пустых метода, которые были переопределены в классе `AuthenticatedKeyExchangeManager`:

1) `namespaces()` возвращает название пространства, под которым передаются заданные в протоколе элементы, код метода представлен в листинге 4;

2) `onStreamPrepared()` задает поведение класса после подготовки XMPP потока.

Листинг 4 – Метод `namespaces()`

```
override func namespaces() -> [String] {
    return [
        "urn:xabber:trust"
    ]
}
```

В методе `onStreamPrepared()` (листинг 5) подготавливается ID устройства, его ключевая пара и доверительный ключ.

Листинг 5 – Метод `onStreamPrepared()`

```
override func onStreamPrepared(_ stream: XMPPStream) {
    super.onStreamPrepared(stream)
    guard let localStore = AccountManager.shared.find(for:
owner)?.omemo.localStore else {
        return
```

```
    }

    self.keyPair = localStore.getIdentityKeyPair()
    self.deviceID = localStore.localDeviceId()

    guard self.keyPair != nil,
          self.deviceID != nil else {
        return
    }

    let publicKey = Array(self.keyPair!.publicKey.dropFirst())
    let fingerprint = publicKey.toHexString()

    self.trustedKey = (String(self.deviceID!) + "::" + fingerprint).bytes
}
```

`AuthenticatedKeyExchangeManager` содержит метод генерации случайной байтовой последовательности устройства `generateByteSequence()` (листинг 6). Эта байтовая последовательность далее используется в алгоритме протокола, при обмене пользователей хэшем.

Листинг 6 – Метод `generateByteSequence()`

```
private final func generateByteSequence() -> [UInt8] {
    var bytes = [UInt8](repeating: 0, count: 32)
    let status = SecRandomCopyBytes(kSecRandomDefault, bytes.count, &bytes)
    if status == errSecSuccess {
        self.byteSequence = bytes
    } else {
        DDLogDebug("AuthenticationKeyExchangeManager: \(#function)")
        fatalError()
    }
}
```

Цифробуквенный шестизначный код генерируется рандомно для каждой сессии аутентификации пользователей и верификации их устройств. Этим кодом пользователи обмениваются вне установленного XMPP канала (лучше лично или по звонку).

Метод `calculateSharedKey()` (листинг 7) класса рассчитывает общий ключ двух устройств, начавших сессию верификации и аутентификации, используя приватный ключ пользователя устройства и публичный ключ его оппонента. На обеих сторонах ключ получается одинаковым. Этот ключ используется для создания общего симметричного ключа шифрования двух устройств.

Листинг 7 – Метод `calculateSharedKey()`

```
internal final func calculateSharedKey(jid: String, deviceId: Int) ->
[UInt8] {
    let keyPair = Curve25519.load(fromPublicKey: self.publicKey, andPri-
vateKey: self.privateKey)
    let opponentPublicKey = getUsersPublicKey(jid: jid, deviceId: de-
viceId)

    let sharedKey = Array(Curve25519.generateSharedSecret(fromPublicKey:
Data(opponentPublicKey), andKeyPair: keyPair))

    return sharedKey
}
```

Метод `calculateEncryptionKey()` (листинг 8) вычисляет общий симметричный ключ шифрования. Для этого он обращается в базу данных за информацией о коде сессии верификации, а потом вычисляет хэш SHA256 от строки, состоящей из общего ключа и хэша от секретного кода.

Листинг 8 – Метод `calculateEncryptionKey()`

```
private final func calculateEncryptionKey(jid: String, sid: String,
sharedKey: [UInt8]) -> [UInt8] {
    var code: String = ""
    do {
        let realm = try WRealm.safe()
        let instance = realm.object(ofType: VerificationSession-
StorageItem.self, forPrimaryKey: VerificationSessionStorageItem.genPri-
mary(owner: self.owner,sid: sid))
        code = instance!.code
    } catch {
        DDLogDebug("AuthenticatedKeyExchange \(#function). \(error.local-
izedDescription)")
    }
    let stringToHash = sharedKey + Array(SHA256.hash(data:
(code.bytes)).makeIterator())
    let encryptionKey = Array(SHA256.hash(data: stringToHash).makeItera-
tor())
    return encryptionKey
}
```

Метод `getMessageChildsForVerififcationRequest()` (листинг 9) класса формирует список дочерних XML-элементов сообщения, которое отправляется оппоненту пользователя в виде запроса на аутентификацию и верификацию устройств. Для запроса, это элемент <`authenticated-key-exchange`>, который содержит дочерний элемент <`verification-start`>. XML-код элемента представлен на рисунке 5.

Листинг 9 – Метод `getMessageChildsForVerififcationRequest()`

```
func getMessageChildsForVerififcationRequest() -> [DDXMLElement] {
    let authenticationKeyExchange = DDXMLElement(name: "authenticated-key-
exchange", xmlns: getPrimaryNamespace())
    authenticationKeyExchange.addAttribute(withName: "sid", stringValue:
UUID().uuidString)

    let verificationStart = DDXMLElement(name: "verification-start")
    verificationStart.addAttribute(withName: "device-id", intValue:
Int32(self.deviceID!))

    if deviceId != nil {
        verificationStart.addAttribute(withName: "to-device-id",
stringValue: deviceId!)
    }

    authenticationKeyExchange.addChild(verificationStart)

    return [authenticationKeyExchange]
}
```

```
<message xml:lang="en" to="bob@xmppdev01.xabber.com" from
    ="bob@xmppdev01.xabber.com/B1" type="headline" id="
    :qprqp0sZ0BvKFKKIZ-ng">
    <authenticated-key-exchange
        xmlns="urn:xabber:trust" sid="234a634f-fc5f-4f71-8b33
            -890ea26b721a" timestamp="1712900258.505384">
        <verification-accepted device-id="350286867" />
    </authenticated-key-exchange>
</message>
```

Рисунок 5 – XML-элемент запроса на верификацию

Метод `getMessageChildsForAcceptVerificationRequest()` (листинг 10) формирует список дочерних XML-элементов сообщения, которое отправляется оппоненту после подтверждения запроса на аутентификацию и верификацию устройств. Это элемент <`authenticated-key-exchange`>, который содержит дочерний элемент <`verification-accepted`> и дочерний элемент <`salt`>, в котором хранится зашифрованная байтовая последовательность пользователя. XML-код элемента представлен на рисунке 6.

Листинг 10 – Метод `getMessageChildsForAcceptVerificationRequest()`

```
func getMessageChildsForAcceptVerificationRequest(sid: String, encrypt-
edByteSequence: String, iv: String) -> [DDXMLElement] {
    let authenticatedKeyExchange = DDXMLElement(name: "authenticated-key-
exchange", xmlns: getPrimaryNamespace())
    authenticatedKeyExchange.addAttribute(withName: "sid", stringValue:
sid)
```

```
    authenticatedKeyExchange.addAttribute(withName: "timestamp",
stringValue: String(Date().timeIntervalSince1970))

    let verificationAccept = DDXMLElement(name: "verification-accepted")
    verificationAccept.addAttribute(withName: "device-id", intValue:
Int32(self.deviceID!))

    let salt = DDXMLElement(name: "salt")
    salt.addChild(DDXMLElement(name: "ciphertext", objectValue: encrypt-
edByteSequence))
    salt.addChild(DDXMLElement(name: "iv", objectValue: iv))

    authenticatedKeyExchange.addChild(verificationAccept)
    authenticatedKeyExchange.addChild(salt)

    return [authenticatedKeyExchange]
}
```

```
<message xml:lang="en" to="alice@xmppdev01.xabber.com/A1" from=
"bob@xmppdev01.xabber.com/B1" type="chat" id=":qprqp0sZ0BvKFKKIZ-ng">
    <authenticated-key-exchange
        xmlns="urn:xabber:trust" sid="234a634f-fc5f-4f71-8b33-890ea26b721a">
        <verification-accepted device-id="350286867" />
        <salt>
            <ciphertext>
WjBGQlFVRkJRbXg2VEVwRlNXOXRha041UWpZMUxUQTNVVEJSUWtaWFdqUTNRa053V1ZkdVZtcEdVMlp
IY3pRNVNtbDRRMFUyVmtjelREQTVRM1EzV25kbVpsWlZWa3BLTTJ0RFQweGFTRVF3VDIxM1VXNTNUM0
U1VjJ3NGRsaGhTMUUxUmsxNk0zcHpXR0V4YlhKcFYzbHFkRmhEVkRKUU9IbDJNV3c1WmtWM1F6Tm9ab
VJuVTFVeGNrWT0=
            </ciphertext>
            <iv>SW9takN5QjY1KzA3UTBRQkZXWjQ3QT09</iv>
        </salt>
    </authenticated-key-exchange>
</message>
```

Рисунок 6 – XML-элемент подтверждения запроса на верификацию

Метод `didReceivedVerificationMessage` (листинг 1 в приложении А) обрабатывает входящие сообщения о верификации. По указанному в контейнере `<authenticated-key-exchange>` идентификатору сессии, менеджер находит сессию с этим же идентификатором в хранилище сессий верификации `VerificationSessionStorageItem`. В зависимости от этапа сессии, менеджер выполняет определенное действие по алгоритму протокола.

Запись устройства в список доверенных устройств реализован в методе `writeTrustedDevice()` (листинг 11). Менеджер находит устройство с определенным идентификатором в хранилище `SignalDeviceStorageItem` и перезаписывает его состояние на доверенное.

Листинг 11 – Метод `writeTrustedDevice()`

```
func writeTrustedDevice(jid: String, deviceId: Int) {
    do {
        let realm = try WRealm.safe()
        if let instance = realm.object(ofType: SignalDeviceStor-
ageItem.self, forPrimaryKey: SignalDeviceStorageItem.genPrimary(owner:
self.owner, jid: jid, deviceId: deviceId)) {
            try realm.write {
                instance.trustDate = Date()
                instance.state = .trusted
            }
        }
    } catch {
        DDLogDebug("AuthenticatedKeyExchangeManager: \(#function). \(er-
ror.localizedDescription)")
    }
}
```

Метод `encrypt()` (листинг 12) зашифровывает переданные в функцию данные. Возвращает шифротекст и вектор инициализации, используемый для шифрования и необходимый для расшифровки данных.

Листинг 12 – Метод `encrypt()`

```
func encrypt(data: Array<UInt8>) -> (encrypted: [UInt8], iv: [UInt8]) {
    var iv = [UInt8](repeating: 0, count: 16)
    let status = SecRandomCopyBytes(kSecRandomDefault, iv.count, &iv)
    guard status == errSecSuccess else {
        DDLogDebug("AuthenticationKeyExchangeManager: \(#function)")
        fatalError()
    }

    let aes = try! AES(key: self.encryptionKey!, blockMode: CBC(iv: iv))
    let encrypted = try! aes.encrypt(data)

    return (encrypted: encrypted, iv: iv)
}
```

Метод `decrypt()` (листинг 13) расшифровывает переданный шифротекст переданным вектором инициализации.

Листинг 13 – Метод `decrypt()`

```
func decrypt(ciphertext: [UInt8], iv: [UInt8]) -> [UInt8] {
    let aes = try! AES(key: self.encryptionKey!, blockMode: CBC(iv: iv))
    let decrypted = try! aes.decrypt(ciphertext)

    return decrypted
}
```

Скриншоты пользовательского интерфейса верификации устройств показаны в приложении Б на рисунках 2–4.

3.3. Реализация хранилища сессий верификаций

Для хранения сессий верификаций в базе данных бал реализован класс `VerificationSessionStorageItem`. Он содержит динамические поля первичного ключа (идентификатор сессии), JID владельца используемого устройства, ID используемого устройства, JID контакта, с которым проходит сессия верификации, ID устройства контакта, случайная байтовая последовательность, созданная для сессии верификации, случайная байтовая последовательность контакта, секретный код, статус сессии, идентификатор сессии, и временную метку сессии (листинг 14).

Листинг 14 – Поля хранилища сессий верификации

```
@objc dynamic var primary: String = ""
@objc dynamic var owner: String = ""
@objc dynamic var myDeviceId: Int = 0
@objc dynamic var jid: String = ""
@objc dynamic var fullJID: String = ""
@objc dynamic var opponentDeviceId: Int = 0
@objc dynamic var byteSequence: String = ""
@objc dynamic var opponentByteSequence: String = ""
@objc dynamic var code: String = ""
@objc dynamic var state_: String = VerififcationState.none.rawValue
@objc dynamic var sid: String = ""
@objc dynamic var opponentByteSequenceEncrypted: String = ""
@objc dynamic var opponentByteSequenceIv: String = ""
@objc dynamic var timestamp: String = ""
```

Статус сессии верификации определяется элементами перечислителя `VerificationState` (листинг 15). Каждый элемент перечислителя определяет этап, на котором находится сессия верификации.

Листинг 15 – Перечислитель `VerificationState`

```
enum VerififcationState: String {
    case none = "none"
    case sentRequest = "sent_request"
    case receivedRequest = "received_request"
    case acceptedRequest = "accepted_request"
    case receivedRequestAccept = "received_request_accept"
    case hashSentToOpponent = "hash_sent_to_opponent"
    case hashSentToInitiator = "hash_sent_to_initiator"
    case trusted = "trusted"
    case rejected = "rejected"
    case failed = "failed"
}
```

3.4. Реализация протокола передачи информации о доверенных устройствах

Для обмена информацией о доверенных устройствах был реализован менеджер `TrustSharingManager`. Он представляет собой класс, который наследуется от абстрактного класса всех менеджеров `AbstractXMPP-Manager`.

В методе `sendNotificationWithContactsDevices()` реализовано отправление списка доверенных устройств контактов пользователя на указанный JID. Код метода приведен в приложении А листинге 2. В списке устройств, хранящихся в хранилище, ищутся все устройства со статусом «доверено», владельцем которых не является пользователь (т. е. все доверенные устройства, кроме своих). Список этих устройств подписывается и вместо с подписью отправляется на другое устройство.

Обработчик сообщений, отправляющихся методом `sendNotificationWithContactsDevices()` (листинг 3 приложения А), реализован в методе `didReceivedTrustedSharingMessage()`. Проверяется подпись сообщения и статус устройства, от которого пришло сообщение. Если подпись верифицирована, а устройство является доверенным, то все устройства из списка записываются как доверенные.

Метод `publicOwnTrustedDevices()` реализует публикацию своих же доверенных устройств пользователя (листинг 4 приложения А). В хранилище устройств ищутся другие устройства пользователя со статусом «доверено». Этот список устройств подписывается и вместе с подписью публикуется на узле «urn:xmpp:trustsharing:0:items».

Для получения информации о том, каким устройствам пользователя доверяют устройства, принадлежащие этому же пользователю, используется метод `getUserTrustedDevices()` (листинг 16). Запрашиваются публикации нужного пользователя на узел «urn:xmpp:trustsharing:0:items». Можно указать ID определенного устройства, если требуется список только его доверенных устройств.

Листинг 16 – Метод `getUserTrustedDevices()`

```
func getUserTrustedDevices(jid: XMPPJID, deviceId: String? = nil) {
    let items = DDXMLElement(name: "items")
    items.addAttribute(withName: "node", stringValue: self.node)

    if deviceId != nil {
        let item = DDXMLElement(name: "item")
        item.addAttribute(withName: "id", stringValue: deviceId!)
        items.addChild(item)
    }

    let pubsub = DDXMLElement(name: "pubsub", xmlns: "http://jab-
ber.org/protocol/pubsub")
    pubsub.addChild(items)

    let iq = XMPPIQ(iqType: .get, to: jid, child: pubsub)

    AccountManager.shared.find(for: self.owner)?.action({ user, stream in
        stream.send(iq)
    })
}
```

Обработчик ответов на запросы из метода выше реализован в методе `read()` (листинг 5 приложения А). Для каждого публикующего устройства из публикации определяется статус. Если устройство является доверенным, то проверяется подпись публикации устройства. В случае успешной верификации подписи, устройства из списка публикации отмечаются как доверенные. Таким образом, устройство доверяет всем доверенным устройствам публикующего устройства, если оно является доверенным.

3.5. Тестирование работы протокола с другими клиентами

В ходе данного тестирования проверялось соответствие приложения функциональным требованиям. Все тесты были успешно пройдены. Ниже (таблица 1) представлены результаты тестирования.

Таблица 1 – Тестирование работы протокола на iOS-клиенте с другими клиентами

№	Название теста	Шаги	Ожидаемый результат	Пройден ли тест
1	Отправление запроса на верификацию	Отправить запрос на верификацию пользователю, подключенному с другого клиента Xabber.	Устройства оппонента, которому отправлен запрос на верификацию, получили его и отобразили запрос пользователю устройств.	Да

Окончание таблицы 1

№	Название теста	Шаги	Ожидаемый результат	Пройдели тест
2	Подтверждение запроса на верификацию	Принять запрос на верификацию, полученный от устройства с другого клиента	Подтверждение отправилось устройству оппонента, на экране устройства пользователя отобразился шестизначный цифробуквенный код.	Да
3	Обмен подлинной информацией	1. Отправить оппоненту запрос на верификацию. 2. Дождаться подтверждения запроса. Ввести код, полученный от оппонента.	Сессия верификации завершилась успешно, информация об этом появилась на устройствах пользователя и оппонента.	Да
4	Обмен измененной информацией	1. Отправить оппоненту запрос на верификацию. 2. Дождаться подтверждения запроса. Ввести код, отличающийся от кода, полученного от оппонента.	Сессия верификации завершилась провалом, информация об этом появилась на устройствах пользователя и оппонента.	Да

Вывод по третьей главе

В этой главе были описаны используемые инструменты, реализация прототипа работы протокола на языке Python, реализация самого протокола в приложении Xabber на клиенте iOS, создание модели для хранения всех сессий верификаций на устройстве, реализация дополнительного протокола, по которому устройства обмениваются списками доверенных устройств.

ЗАКЛЮЧЕНИЕ

В рамках данной работы был разработан протокол XEP-TRUST: Authenticated Key Exchange, который позволяет пользователям аутентифицировать друг друга и верифицировать свои устройства между собой. Разработанный протокол минимизирует участие пользователей в этом процессе. Теперь им требуется только обменяться секретным кодом. При этом были решены нижеизложенные задачи.

1. Произведен обзор литературы и существующих алгоритмов по предметной области.
2. Разработан алгоритм, используемый в протоколе.
3. Подготовлен текст документации протокола XEP.
4. Разработан прототип реализации протокола, демонстрирующий процесс аутентификации пользователей и верификации их устройств по разработанному протоколу.
5. Реализован протокол аутентификации и верификации устройств пользователей на iOS-клиенте Xabber.
6. Реализован протокол в веб-клиенте Xabber другими разработчиками.
7. Протестирована работа протокола на iOS-клиенте с веб-клиентом приложения.

ЛИТЕРАТУРА

1. Что такое идентификатор: определение и применение. [Электронный ресурс] URL: https://businessman.ru/2023-chto-takoe-identifikator-opredelenie-i-primenenie.html (дата обращения: 26.04.2024 г.).

2. Straub A., Gultsch D., Henkes T.,Herberth K., Schaub P., Wißfeld M. XEP-0384: OMEMO Encryption. // XMPP Standards Foundation, 2015. – 26 с.

3. Что такое аутентификация. [Электронный ресурс] URL: https://iaassaaspaas.ru/terminologiya/chto-takoe-autentifikatsiya (дата обращения: 26.04.2024 г.).

4. Что такое верификация: простыми словами и где она применяется. [Электронный ресурс] URL: https://pravdk.ru/chto-takoe-verifikaciya-prostymi-slovami-i-gde-ona-primenyaetsya/ (дата обращения: 26.04.2024 г.).

5. Асимметричное шифрование на практике. [Электронный ресурс] URL: https://habr.com/ru/articles/449552/ (дата обращения: 21.05.2024 г.).

6. Как устроен AES. [Электронный ресурс] URL: https://habr.com/ru/articles/112733/ (дата обращения: 08.02.2024 г.).

7. Keskin M. XEP-0450: Automatic Trust Management (ATM). // XMPP Standards Foundation, 2020. – 10 с.

8. Что такое компрометация данных и как ей защититься. [Электронный ресурс] URL: https://helpdoma.ru/faq/cto-takoe-komprometaciya-dannyx-i-kak-ei-zashhititsya (дата обращения: 26.04.2024 г.).

9. Что такое хэш в криптографии? Как работает хеширование? [Электронный ресурс] URL: https://informationsecurityasia.com/ru/what-is-a-hash/ (дата обращения: 26.04.2024 г.).

10. С позиции энтропии, как шифрование влияет на сообщение? [Электронный ресурс] URL: https://bigdevops.ru/article/s-pozitsii-entropii-kak-shifrovanie-vliyaet-na-soobschenie (дата обращения: 16.05.2024 г.).

11. Kaes C. XEP-0029: Definition of Jabber Identifiers (JIDs). // XMPP Standards Foundation, 2002. – 3 с.

12. Saint-Andre P. RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core. // Internet Engineering Task Force (IETF), 2011. – 210 с.

13. Saint-Andre P., Cridland D., Meijer R. XEP-0001: XMPP Extension Protocols. // XMPP Standards Foundation, 2001. – 16 с.

14. What Is End-to-End Encryption, and Why Does It Matter? [Электронный ресурс] URL: https://www.howtogeek.com/711656/what-is-end-to-end-encryption-and-why-does-it-matter/ (дата обращения: 26.05.2024 г.)

15. Nakov S., Stefanov M., Shideroff M. Practical Cryptography for Developers. Digital Signatures. // SoftUni (Software University), 2018. [Электронный ресурс] URL: https://cryptobook.nakov.com/digital-signatures (дата обращения: 10.02.2024 г.).

16. Saint-Andre P. RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. // Internet Engineering Task Force (IETF), 2011. – 113 с.

17. Whited S. XEP-0364: Current Off-the-Record Messaging Usage. // XMPP Standards Foundation, 2015. – 6 с.

18. Muldowney T. XEP-0027: Current Jabber OpenPGP Usage. // XMPP Standards Foundation, 2002. – 4 с.

19. Протокол TLS: что это, зачем нужен и как работает. [Электронный ресурс] URL: https://skillbox.ru/media/code/protokol-tls-chto-eto-zachem-nuzhen-i-kak-rabotaet/#stk-1 (дата обращения: 26.05.2024 г.).

20. Hildebrand J., Miller M., Lukas G. XEP-0280: Message Carbons. // XMPP Standards Foundation, 2021. – 14 с.

21. Как работает обмен ключами в протоколе Диффи-Хеллмана. [Электронный ресурс] URL: https://tproger.ru/translations/diffie-hellman-key-exchange-explained (дата обращения: 16.05.2024 г.).

22. Millard P., Saint-Andre P., Meijer R. XEP-0060: Publish-Subscribe. // XMPP Standards Foundation, 2023. – 179 с.

23. XMPPFramework. Extensible Messaging and Presence Protocol Framework. [Электронный ресурс] URL: https://github.com/robbiehanson/XMPPFramework (дата обращения: 09.02.2024 г.).

24. Curve25519, EdDSA и Poly1305: Три обделенных вниманием криптопримитива. [Электронный ресурс] URL: https://habr.com/ru/articles/247873/ (дата обращения: 03.01.2024 г.).

ПРИЛОЖЕНИЯ

Приложение А. Код реализации протокола

Листинг 1 – Код метода `didReceivedVerificationMessage` класса `AuthenticatedKeyExchange`

```
func didReceivedVerificationMessage(_ message: XMPPMessage) -> Bool {
    if isArchivedMessage(message) {
        return false
    } else if isCarbonCopy(message) {
        return false
    } else if isCarbonForwarded(message) {
        return false
    } else {
        guard let authenticatedKeyExchange = message.element(forName: "au-
thenticated-key-exchange", xmlns: getPrimaryNamespace()),
                let jid = message.from,
                let sid = authenticatedKeyExchange.attribute-
StringValue(forName: "sid") else {
            return false
        }

        var title = ""

        if authenticatedKeyExchange.element(forName: "verification-start")
!= nil {
            let timestamp = Date().timeIntervalSince1970
            do {
                let realm = try WRealm.safe()
                if realm.object(ofType: VerificationSession-
StorageItem.self, forPrimaryKey: VerificationSessionStorageItem.genPri-
mary(owner: self.owner, sid: sid)) != nil {
                    return true
                }
                guard let verificationStart = authenticatedKeyExchange.ele-
ment(forName: "verification-start"),
                        let opponentDeviceID = Int((verificationStart.at-
tributeStringValue(forName: "device-id"))!) else {
                    DDLogDebug("Opponent device ID is not specified")
                    return true
                }

                let predicate = NSPredicate(format: "owner == %@ AND myDe-
viceId == %@ AND opponentDeviceId == %@", argumentArray: [self.owner,
self.deviceID!, opponentDeviceID])
                let oldInstances = realm.objects(VerificationSession-
StorageItem.self).filter(predicate)
                if !oldInstances.isEmpty {
                    try realm.write {
                        realm.delete(oldInstances)
                    }
                }

                let instance = VerificationSessionStorageItem()
                instance.owner = self.owner
                instance.myDeviceId = self.deviceID!
                instance.jid = jid.bare
                instance.fullJID = jid.full
                instance.sid = sid
                instance.opponentDeviceId = opponentDeviceID
                instance.state = .receivedRequest
```

```
                instance.primary = VerificationSessionStorageItem.genPri-
mary(owner: self.owner, sid: sid)
                instance.timestamp = String(timestamp)
                try realm.write {
                    realm.add(instance)
                }
            } catch {
                DDLogDebug("AuthenticatedKeyExchange \(#function). \(er-
ror.localizedDescription)")
                fatalError()
            }
            title = "Verification request received"
        } else if authenticatedKeyExchange.element(forName: "verification-
accepted") != nil {
            let byteSequence = self.generateByteSequence()
            let timestamp = Date().timeIntervalSince1970

            do {
                let realm = try WRealm.safe()
                guard let instance = realm.object(ofType: Verifica-
tionSessionStorageItem.self, forPrimaryKey: VerificationSession-
StorageItem.genPrimary(owner: self.owner, sid: sid)) else {
                    return true
                }
                if instance.state != .sentRequest {
                    return true
                }

                guard let verificationAccepted = authenticat-
edKeyExchange.element(forName: "verification-accepted"),
                      let opponentDeviceID = Int((verificationAc-
cepted.attributeStringValue(forName: "device-id"))!),
                      let saltEncrypted = authenticatedKeyExchange.ele-
ment(forName: "salt")?.element(forName: "ciphertext")?.stringValue,
                      let saltIv = authenticatedKeyExchange.element(for-
Name: "salt")?.element(forName: "iv")?.stringValue else {
                    return true
                }

                try realm.write {
                    instance.jid = jid.bare
                    instance.fullJID = jid.full
                    instance.opponentDeviceId = opponentDeviceID
                    instance.byteSequence = byteSequence.toBase64()
                    instance.state = .receivedRequestAccept
                    instance.opponentByteSequenceEncrypted = saltEncrypted
                    instance.opponentByteSequenceIv = saltIv
                    instance.timestamp = String(timestamp)
                }
            } catch {
                DDLogDebug("AuthenticatedKeyExchange \(#function). \(er-
ror.localizedDescription)")
                fatalError()
            }

            title = "Verification request accepted"
        } else if authenticatedKeyExchange.element(forName: "hash") != nil
&& authenticatedKeyExchange.element(forName: "salt") != nil {
            guard let hashEncrypted = authenticatedKeyExchange.element(for-
Name: "hash"),
```

```
                    let byteSequenceEncrypted = authenticat-
edKeyExchange.element(forName: "salt") else {
                return false
            }

            var deviceId: Int = 0

            do {
                let realm = try WRealm.safe()
                guard let instance = realm.object(ofType: Verifica-
tionSessionStorageItem.self, forPrimaryKey: VerificationSession-
StorageItem.genPrimary(owner: self.owner, sid: sid)) else {
                    return true
                }
                if instance.state != .acceptedRequest {
                    return true
                }

                deviceId = instance.opponentDeviceId

                try realm.write {
                    instance.state = .hashSentToInitiator
                    instance.timestamp = String(Date().timeInter-
valSince1970)
                }
            } catch {
                DDLogDebug("AuthenticatedKeyExchange \(#function). \(er-
ror.localizedDescription)")
            }

            if !checkHashFromInitiator(jid: jid.bare, sid: sid, deviceId:
deviceId, hashEncrypted: hashEncrypted, byteSequenceEncrypted: byteSe-
quenceEncrypted) {
                let child = self.getMessageChildsForErrorMessage(sid: sid,
reason: "Hashes didn't match")
                let message = XMPPMessage(messageType: .chat, to: jid, ele-
mentID: UUID().uuidString, child: child)
                AccountManager.shared.find(for: self.owner)?.unsafeAction({
user, stream in
                    stream.send(message)
                })
                do {
                    let realm = try WRealm.safe()
                    guard let instance = realm.object(ofType: Verifica-
tionSessionStorageItem.self, forPrimaryKey: VerificationSession-
StorageItem.genPrimary(owner: self.owner, sid: sid)) else {
                        fatalError()
                    }
                    try realm.write {
                        instance.state = .failed
                    }
                } catch {
                    fatalError()
                }
                self.showNotification(title: jid.bare, owner: self.owner,
body: "Verification failed", sid: sid, timestamp: Date().timeInter-
valSince1970)
                return true
            }
            let hash = self.calculateHashForInitiator(jid: jid.bare, sid:
sid)
```

```
            let encryptedHashResult = self.encrypt(jid: jid.bare, sid: sid,
deviceId: deviceId, data: hash)

            let authenticatedKeyExchangeChild = DDXMLElement(name: "authen-
ticated-key-exchange", xmlns: getPrimaryNamespace())
            authenticatedKeyExchangeChild.addAttribute(withName: "sid",
stringValue: sid)

            let hashChild = DDXMLElement(name: "hash")
            hashChild.addAttribute(withName: "algo", stringValue: "sha-
256")
            hashChild.addChild(DDXMLElement(name: "ciphertext",
stringValue: encryptedHashResult.encrypted.toBase64()))
            hashChild.addChild(DDXMLElement(name: "iv", stringValue: en-
cryptedHashResult.iv.toBase64()))

            authenticatedKeyExchangeChild.addChild(hashChild)

            let message = XMPPMessage(messageType: .chat, to: jid, elemen-
tID: UUID().uuidString, child: authenticatedKeyExchangeChild)
            AccountManager.shared.find(for: self.owner)?.unsafeAction({
user, stream in
                stream.send(message)
            })

            return true
        } else if authenticatedKeyExchange.element(forName: "hash") != nil
{
            guard let hashEncrypted = authenticatedKeyExchange.element(for-
Name: "hash") else {
                return false
            }

            var deviceId: Int = 0
            var byteSequence: [UInt8] = []
            var opponentByteSequence: [UInt8] = []
            var code: String = ""

            do {
                let realm = try WRealm.safe()
                let instance = realm.object(ofType: VerificationSession-
StorageItem.self, forPrimaryKey: VerificationSessionStorageItem.genPri-
mary(owner: self.owner, sid: sid))
                if instance?.state != .hashSentToOpponent {
                    return true
                }
                deviceId = instance!.opponentDeviceId
                code = instance!.code
                byteSequence = try instance!.byteSequence.base64decoded()
                opponentByteSequence = try instance!.opponentByteSe-
quence.base64decoded()

                try realm.write {
                    instance?.state = .hashSentToInitiator
                    instance?.timestamp = String(Date().timeInter-
valSince1970)
                }

            } catch {
                DDLogDebug("AuthenticatedKeyExchangeManager: \(#function).
\(error.localizedDescription)")
```

```
                fatalError()
            }

            let publicKey = self.getUsersPublicKey(jid: jid.bare, deviceId:
deviceId)
            let fingerprint = publicKey.toHexString()

            let hash = self.decryptElementFromXML(jid: jid.bare,
                                                  sid: sid,
                                                  deviceId: deviceId,
                                                  encryptedXML:
hashEncrypted)

            let opponentTrustedKey = String(deviceId) + "::" + fingerprint

            let stringToHash = opponentTrustedKey.bytes + Array(code.utf8)
+ opponentByteSequence + byteSequence
            let myHash = Array(SHA256.hash(data: stringToHash).makeItera-
tor())

            do {
                let realm = try WRealm.safe()
                let instance = realm.object(ofType: VerificationSession-
StorageItem.self, forPrimaryKey: VerificationSessionStorageItem.genPri-
mary(owner: self.owner, sid: sid))
                if hash != myHash {
                    sendErrorMessage(fullJID: jid, sid: sid, reason:
"Hashes didn't match")
                    self.showNotification(title: jid.bare, owner:
self.owner, body: "Verification failed", sid: sid, timestamp:
Date().timeIntervalSince1970)
                    try realm.write {
                        instance?.state = .failed
                    }
                    return true
                }
                try realm.write {
                    instance?.state = .trusted
                }
            } catch {
                DDLogDebug("AuthenticatedKeyExchangeManager: \(#function).
\(error.localizedDescription)")
                fatalError()
            }

            self.sendSuccessfulVerificationMessage(fullJID: jid, sid: sid)

            if self.owner == jid.bare {
                guard let trustSharingManager = AccountMan-
ager.shared.find(for: self.owner)?.trustSharingManager else {
                    fatalError()
                }
                trustSharingManager.sendMessageWithContactsDevices(oppo-
nentFullJid: jid, deviceId: self.deviceID!, opponentDeviceId: deviceId)
            }

            title = "Verification completed successfully"
        } else if authenticatedKeyExchange.element(forName: "verification-
successful") != nil {
            var deviceId: Int = 0
            do {
```

```
                let realm = try WRealm.safe()
                let instance = realm.object(ofType: VerificationSession-
StorageItem.self, forPrimaryKey: VerificationSessionStorageItem.genPri-
mary(owner: self.owner, sid: sid))
                if instance?.state != .hashSentToInitiator {
                    return true
                }
                deviceId = instance!.opponentDeviceId

                try realm.write {
                    instance?.state = .trusted
                }
            } catch {
                fatalError()
            }

            if self.owner == jid.bare {
                guard let trustSharingManager = AccountMan-
ager.shared.find(for: self.owner)?.trustSharingManager else {
                    fatalError()
                }
                trustSharingManager.sendMessageWithContactsDevices(oppo-
nentFullJid: jid, deviceId: self.deviceID!, opponentDeviceId: deviceId)
            }

            title = "Verification completed successfully"
        } else if authenticatedKeyExchange.element(forName: "verification-
rejected") != nil {
            do {
                let realm = try WRealm.safe()
                guard let instance = realm.object(ofType: Verifica-
tionSessionStorageItem.self, forPrimaryKey: VerificationSession-
StorageItem.genPrimary(owner: self.owner, sid: sid)) else {
                    return true
                }
                try realm.write {
                    instance.state = .rejected
                }
            } catch {
                fatalError()
            }
            title = "Verification rejected"
        } else if authenticatedKeyExchange.element(forName: "verification-
failed") != nil {
            do {
                let realm = try WRealm.safe()
                guard let instance = realm.object(ofType: Verifica-
tionSessionStorageItem.self, forPrimaryKey: VerificationSession-
StorageItem.genPrimary(owner: self.owner, sid: sid)) else {
                    return true
                }
                try realm.write {
                    instance.state = .failed
                }
            } catch {
                fatalError()
            }
            title = "Verification failed"
        }
        self.showNotification(title: jid.bare, owner: self.owner, body: ti-
tle, sid: sid, timestamp: Date().timeIntervalSince1970)
```

```
    }
    return true
}
```

Листинг 2 – Код метода `sendNotificationWithContactsDevices()` класса `TrustSharingManager`

```
func sendNotificationWithContactsDevices(opponentFullJid: XMPPJID, de-
viceId: Int) {
    let share = DDXMLElement(name: "share", xmlns: self.getPrimaryName-
space())
    share.addAttribute(withName: "usage", stringValue: "urn:xmpp:omemo:2")

    do {
        let realm = try WRealm.safe()
        let predicate = NSPredicate(format: "owner == %@ AND jid != %@ AND
state_ == %@", argumentArray: [self.owner, self.owner, "trusted"])
        let instances = realm.objects(SignalDeviceStorageItem.self).fil-
ter(predicate)
        if instances.isEmpty {
            return
        }
        var jids: [String] = []
        for instance in instances {
            if !jids.contains(where: { $0 == instance.jid }) {
                jids.append(instance.jid)
            }
        }
        for jid in jids {
            let trustedItems = DDXMLElement(name: "trusted-items")
            trustedItems.addAttribute(withName: "owner", stringValue: jid)
            trustedItems.addAttribute(withName: "timestamp", stringValue:
String(Date().timeIntervalSince1970))
            for instance in instances {
                if instance.jid == jid {
                    let trustedKey = String(instance.deviceId) + "::" + in-
stance.fingerprint
                    let trust = DDXMLElement(name: "trust", stringValue:
trustedKey.toBase64())
                    trust.addAttribute(withName: "timestamp", stringValue:
String(instance.trustDate.timeIntervalSince1970))
                    trustedItems.addChild(trust)
                }
            }
            share.addChild(trustedItems)
        }
    } catch {
        fatalError()
    }

    guard let akeManager = AccountManager.shared.find(for:
self.owner)?.akeManager,
          let privateKey = akeManager.keyPair?.privateKey.bytes else {
        fatalError()
    }
    let publicKey = akeManager.getUsersPublicKey(jid: self.owner, deviceId:
deviceId)
    let fingerprint = publicKey.toHexString()
```

```
    let identityXML = DDXMLElement(name: "identity", stringValue: finger-
print)
    identityXML.addAttribute(withName: "id", stringValue: String(deviceId))
    share.addChild(identityXML)

    var stringToHash = ""
    let trustedItemsList = share.elements(forName: "trusted-
items").sorted(by: { $0.attributeStringValue(forName: "timestamp")! >
$1.attributeStringValue(forName: "timestamp")! })

    for item in trustedItemsList {
        stringToHash += item.attribute(forName: "timestamp")!.stringValue!
        let trustsList = item.elements(forName: "trust").sorted(by: {
$0.attributeStringValue(forName: "timestamp")! > $1.attribute-
StringValue(forName: "timestamp")! })
        for trust in trustsList {
            stringToHash += "<" + trust.attributeStringValue(forName:
"timestamp")! + "/" + trust.stringValue!
        }
        stringToHash += "<"
    }

    let keyPair = Curve25519.load(fromPublicKey: akeManager.keyPair?.pub-
licKey, andPrivateKey: akeManager.keyPair?.privateKey)
    let signature = Ed25519.sign(Data(stringToHash.bytes), with: keyPair)

    let signatureXML = DDXMLElement(name: "signature", stringValue: signa-
ture!.base64EncodedString())
    signatureXML.addAttribute(withName: "xmlns", stringValue: self.getPri-
maryNamespace())
    share.addChild(signatureXML)

    let message = XMPPMessage(messageType: .chat, to: opponentFullJid, ele-
mentID: UUID().uuidString, child: share)
    message.addAttribute(withName: "from", stringValue: AccountMan-
ager.shared.find(for: self.owner)!.xmppStream.myJID!.full)

    let iq = akeManager.getNotificationContainer(message: message, notifi-
cationTo: opponentFullJid)

    AccountManager.shared.find(for: self.owner)?.action({ user, stream in
        stream.send(iq)
    })
}
```

Листинг 3 – Код метода `didReceivedTrustedSharingMessage` класса `TrustSharingManager`

```
func didReceivedTrustedSharingMessage(message: XMPPMessage) -> Bool {
    let bareMessage: XMPPMessage
    if isArchivedMessage(message) {
        bareMessage = getArchivedMessageContainer(message)!
    } else if isCarbonCopy(message) {
        return false
    } else if isCarbonForwarded(message) {
        return false
    } else {
        bareMessage = message
    }
```

```
    guard let notify = bareMessage.element(forName: "notify", xmlns: XMP-
PNotificationsManager.xmlns) ?? bareMessage.element(forName: "notifica-
tion", xmlns: XMPPNotificationsManager.xmlns) else {
        return false
    }
    let uniqueMessageId = getUniqueMessageId(bareMessage, owner:
self.owner)
    guard let messageContainer = notify.element(forName: "forwarded")?.ele-
ment(forName: "message") else {
        return false
    }
    guard let share = messageContainer.element(forName: "share", xmlns:
getPrimaryNamespace()),
          let jid = XMPPMessage(from: messageContainer).from,
          let signature = try! share.element(forName: "signa-
ture")?.stringValue?.base64decoded(),
          let identity = share.element(forName: "identity"),
          let fingerprint = identity.stringValue,
          let deviceId = Int(identity.attributeStringValue(forName:
"id")!),
          let akeManager = AccountManager.shared.find(for:
self.owner)?.akeManager else {
        return false
    }

    if jid.full == AccountManager.shared.find(for:
self.owner)!.xmppStream.myJID!.full {
        return true
    }

    let predicate = NSPredicate(format: "owner == %@ AND jid == %@ AND de-
viceId == %@", argumentArray: [self.owner, jid.bare, deviceId])
    do {
        let realm = try WRealm.safe()
        guard let instance = realm.objects(SignalDeviceStor-
ageItem.self).filter(predicate).first else {
            return true
        }
        if instance.state != SignalDeviceStorageItem.TrustState.trusted {
            return true
        }
    } catch {
        fatalError()
    }

    var stringToVerifySignature = ""
    let trustedItemsList = share.elements(forName: "trusted-
items").sorted(by: { $0.attributeStringValue(forName: "timestamp")! >
$1.attributeStringValue(forName: "timestamp")! })

    for item in trustedItemsList {
        stringToVerifySignature += item.attribute(forName:
"timestamp")!.stringValue!
        let trustsList = item.elements(forName: "trust").sorted(by: {
$0.attributeStringValue(forName: "timestamp")! > $1.attribute-
StringValue(forName: "timestamp")! })
        for trust in trustsList {
            stringToVerifySignature += "<" + trust.attribute-
StringValue(forName: "timestamp")! + "/" + trust.stringValue!
        }
        stringToVerifySignature += "<"
```

```
    }
    let userPublicKey = akeManager.getUsersPublicKey(jid: jid.bare, de-
viceId: deviceId)
    if !Ed25519.verifySignature(Data(signature), publicKey:
Data(userPublicKey), data: Data(stringToVerifySignature.bytes)) {
        do {
            let realm = try WRealm.safe()
            guard let instance = realm.objects(VerificationSession-
StorageItem.self).filter(predicate).first else {
                fatalError()
            }
            akeManager.sendErrorMessage(fullJID: jid, sid: instance.sid,
reason: "Error when exchanging trusted devices")
            akeManager.showNotification(title: jid.bare, owner: self.owner,
body: "Verification failed", sid: instance.sid, timestamp: Date().timeIn-
tervalSince1970)
            try realm.write {
                instance.state = .failed
            }
        } catch {
            fatalError()
        }
        return true
    }
    do {
        for item in trustedItemsList {
            let deviceOwner = item.attributeStringValue(forName: "owner")
            let trustsList = item.elements(forName: "trust")
            for trust in trustsList {
                guard let trustKey = try String(bytes:
(trust.stringValue?.base64decoded())!, encoding: .utf8),
                      let itemDeviceId = Int(trustKey.components(separat-
edBy: "::")[0]) else {
                    fatalError()
                }

                let predicate = NSPredicate(format: "owner == %@ AND jid ==
%@ AND deviceId == %@", argumentArray: [self.owner, deviceOwner!,
itemDeviceId])
                let realm = try WRealm.safe()
                guard let instance = realm.objects(SignalDeviceStor-
ageItem.self).filter(predicate).first else {
                    fatalError()
                }
                if instance.state != SignalDeviceStor-
ageItem.TrustState.trusted {
                    akeManager.writeTrustedDevice(jid: deviceOwner ??
XMPPMessage(from: messageContainer).from!.bare, deviceId: itemDeviceId)
                    try realm.write {
                        instance.trustedByDeviceId = String(deviceId)
                    }
                    self.getUserTrustedDevices(jid: XMPPJID(string: device-
Owner!)!, deviceId: String(itemDeviceId))
                }
            }
        }
        return true
    } catch {
        fatalError()
    }
}
```

Листинг 4 – Код метода `publicOwnTrustedDevices()` класса `TrustSharingManager`

```
func publicOwnTrustedDevices(publisherDeviceId: String) {
    let share = DDXMLElement(name: "share", xmlns: self.getPrimaryName-
space())
    share.addAttribute(withName: "usage", stringValue: "urn:xmpp:omemo:2")
    do {
        let realm = try WRealm.safe()
        let predicate = NSPredicate(format: "owner == %@ AND jid == %@ AND
state_ == %@", argumentArray: [self.owner, self.owner, "trusted"])
        let instances = realm.objects(SignalDeviceStorageItem.self).fil-
ter(predicate)
        let trustedItems = DDXMLElement(name: "trusted-items")
        trustedItems.addAttribute(withName: "timestamp", stringValue:
String(Date().timeIntervalSince1970))
        for instance in instances {
            let trustedKey = String(instance.deviceId) + "::" + in-
stance.fingerprint
            let trust = DDXMLElement(name: "trust", stringValue: trust-
edKey.toBase64())
            trust.addAttribute(withName: "timestamp", stringValue:
String(instance.trustDate.timeIntervalSince1970))
            trustedItems.addChild(trust)
        }
        share.addChild(trustedItems)
    } catch {
        fatalError()
    }

    guard let localStore = AccountManager.shared.find(for:
owner)?.omemo.localStore else {
        fatalError()
    }

    let keyPair = localStore.getIdentityKeyPair()
    let publicKey = keyPair.publicKey.dropFirst()

    let fingerprint = publicKey.toHexString()

    let identityXML = DDXMLElement(name: "identity", stringValue: finger-
print)
    identityXML.addAttribute(withName: "id", stringValue: String(publisher-
DeviceId))
    share.addChild(identityXML)

    var stringToHash = ""
    let trustedItemsList = share.elements(forName: "trusted-
items").sorted(by: { $0.attributeStringValue(forName: "timestamp")! >
$1.attributeStringValue(forName: "timestamp")! })

    for item in trustedItemsList {
        stringToHash += item.attribute(forName: "timestamp")!.stringValue!
        let trustsList = item.elements(forName: "trust").sorted(by: {
$0.attributeStringValue(forName: "timestamp")! > $1.attribute-
StringValue(forName: "timestamp")! })
        for trust in trustsList {
            stringToHash += "<" + trust.attributeStringValue(forName:
"timestamp")! + "/" + trust.stringValue!
        }
```

```
        stringToHash += "<"
    }
    let keyPairCurve25519 = Curve25519.load(fromPublicKey: keyPair.pub-
licKey, andPrivateKey: keyPair.privateKey)
    let signature = Ed25519.sign(Data(stringToHash.bytes), with: keyPair-
Curve25519)

    let signatureXML = DDXMLElement(name: "signature", stringValue: signa-
ture!.base64EncodedString())
    signatureXML.addAttribute(withName: "xmlns", stringValue: self.getPri-
maryNamespace())
    share.addChild(signatureXML)

    let item = DDXMLElement(name: "item")
    item.addChild(share)
    item.addAttribute(withName: "id", stringValue: publisherDeviceId)
    let publish = DDXMLElement(name: "publish")
    publish.addAttribute(withName: "node", stringValue: self.node)
    publish.addChild(item)
    let pubsub = DDXMLElement(name: "pubsub", xmlns: "http://jab-
ber.org/protocol/pubsub")
    pubsub.addChild(publish)
    let iq = XMPPIQ(iqType: .set, child: pubsub)

    AccountManager.shared.find(for: self.owner)?.action({ user, stream in
        stream.send(iq)
    })
}
```

Листинг 5 – Код обработчика IQ-станз с публикациями о доверенных устройствах

```
override func read(withIQ iq: XMPPIQ) -> Bool {
    guard let jid = iq.from,
          let pubsub = iq.element(forName: "pubsub", xmlns: "http://jab-
ber.org/protocol/pubsub"),
          let items = pubsub.element(forName: "items") else {
        return false
    }
    let itemList = items.elements(forName: "item")
    if itemList.isEmpty {
        return false
    }
    var isPublicationNeeded = false
    for item in itemList {
        guard let publisherDeviceId = item.attributeStringValue(forName:
"id"),
              let share = item.element(forName: "share"),
              let signature = try! share.element(forName: "signa-
ture")?.stringValue?.base64decoded(),
              let akeManager = AccountManager.shared.find(for:
self.owner)?.akeManager else {
            return false
        }

        let predicateForDevices = NSPredicate(format: "owner == %@ AND jid
== %@ AND deviceId == %@", argumentArray: [self.owner, jid.bare, Int(pub-
lisherDeviceId)!])
        do {
```

```
            let realm = try WRealm.safe()
            guard let instance = realm.objects(SignalDeviceStor-
ageItem.self).filter(predicateForDevices).first else {
                continue
            }
            if instance.state != SignalDeviceStorageItem.TrustState.trusted
{
                continue
            }
        } catch {
            fatalError()
        }

        var stringToVerifySignature = ""
        let trustedItemsList = share.elements(forName: "trusted-
items").sorted(by: { $0.attributeStringValue(forName: "timestamp")! >
$1.attributeStringValue(forName: "timestamp")! })

        for item in trustedItemsList {
            stringToVerifySignature += item.attribute(forName:
"timestamp")!.stringValue!
            let trustsList = item.elements(forName: "trust").sorted(by: {
$0.attributeStringValue(forName: "timestamp")! > $1.attribute-
StringValue(forName: "timestamp")! })
            for trust in trustsList {
                stringToVerifySignature += "<" + trust.attribute-
StringValue(forName: "timestamp")! + "/" + trust.stringValue!
            }
            stringToVerifySignature += "<"
        }

        let userPublicKey = akeManager.getUsersPublicKey(jid: jid.bare, de-
viceId: Int(publisherDeviceId)!)
        if !Ed25519.verifySignature(Data(signature), publicKey:
Data(userPublicKey), data: Data(stringToVerifySignature.bytes)) {
            continue
        }
        do {
            for item in trustedItemsList {
                let trustsList = item.elements(forName: "trust")
                for trust in trustsList {
                    guard let trustKey = try String(bytes:
(trust.stringValue?.base64decoded())!, encoding: .utf8),
                          let deviceId = Int(trustKey.components(separat-
edBy: "::")[0]) else {
                        fatalError()
                    }
                    let predicateForDevices = NSPredicate(format: "owner ==
%@ AND jid == %@ AND deviceId == %@", argumentArray: [self.owner, jid.bare,
deviceId])
                    do {
                        let realm = try WRealm.safe()
                        guard let instance = realm.objects(SignalDeviceS-
torageItem.self).filter(predicateForDevices).first else {
                            continue
                        }
                        if instance.state != SignalDeviceStor-
ageItem.TrustState.trusted {
                            akeManager.writeTrustedDevice(jid: jid.bare,
deviceId: deviceId)
                            try realm.write {
```

Окончание листинга 5 приложения А

```
                                instance.trustedByDeviceId = publisherDe-
viceId
                            }

                            // if the device has trusted its device then it
should publish a new list of trusted devices of the device
                            if self.owner == jid.bare {
                                isPublicationNeeded = true
                            }

                            let item = DDXMLElement(name: "item")
                            item.addAttribute(withName: "id", stringValue:
String(deviceId))
                            let itemsToGet = DDXMLElement(name: "items")
                            itemsToGet.addAttribute(withName: "node",
stringValue: self.node)
                            itemsToGet.addChild(item)
                            let pubsub = DDXMLElement(name: "pubsub",
xmlns: "http://jabber.org/protocol/pubsub")
                            pubsub.addChild(itemsToGet)
                            let iq = XMPPIQ(iqType: .get, to: jid, child:
pubsub)

                            AccountManager.shared.find(for:
self.owner)?.action({ user, stream in
                                stream.send(iq)
                            })
                        }
                    } catch {
                        fatalError()
                    }
                }
            }
            continue
        } catch {
            fatalError()
        }
    }
    if isPublicationNeeded {
        guard let localStore = AccountManager.shared.find(for:
owner)?.omemo.localStore else {
            fatalError()
        }
        self.publicOwnTrustedDevices(publisherDeviceId: String(lo-
calStore.localDeviceId()))
    }
    return true
}
```

Листинг 6 – Код метода `message_received()` прототипа реализации

```
def message_received(self, msg):
    if msg.type_ != aioxmpp.MessageType.CHAT:
        return msg

    # if the message was sent by another device of the same account
    if msg.xep0280_sent is not None:
        if msg.xep0280_sent.forwarded.stanza.to == self.client.local_jid:
            return msg
        if not self.is_passive_mode_enabled:
```

```
            self.received_message_carbons()
        if msg.xep0280_sent.forwarded.stanza.xep0000 is None:
            return msg
        if msg.xep0280_sent.forwarded.stanza.xep0000.verification_ended is
not None:
            result_text = msg.xep0280_sent.forwarded.stanza.xep0000.verifi-
cation_ended.code
            console_string = result_text
            print(console_string)
            self.finish_verification_session()
        return msg
    # if the message was received by another device of the same account
    if msg.xep0280_received is not None:
        if msg.xep0280_received.forwarded.stanza.from_ == self.client.lo-
cal_jid:
            return msg
        if self.is_initiator:
            return msg
        if not self.is_passive_mode_enabled:
            self.received_message_carbons()
        if msg.xep0280_received.forwarded.stanza.xep0000 is None:
            return msg
        if msg.xep0280_received.forwarded.stanza.xep0000.verification_ended
is not None:
            result_text = msg.xep0280_received.for-
warded.stanza.xep0000.verification_ended.code
            console_string = result_text
            print(console_string)
            self.finish_verification_session()
        return msg
    if self.is_passive_mode_enabled:
        return msg

    route = f"<{msg.from_.localpart} -> {msg.to.localpart}>: "

    if msg.xep0000 is None:
        return msg

    # if the message is response to the verification request
    if msg.xep0000.verification_accepted is not None:
        console_string = "Signed salt"
        print(f'{route}{console_string}')
        opponents_jid = str(msg.from_).split("_")[0]
        self.opponent = opponent.Opponent(aioxmpp.JID.fromstr(oppo-
nents_jid))
        self.opponent.device_id = msg.xep0000.verification_accepted.de-
vice_id
        asyncio.ensure_future(self.get_opponent_pub-
key(str(msg.from_.bare())))
        asyncio.ensure_future(self.verification_request_accepted(msg,
route))
        return msg

    # if the message from the opponent containing the encrypted hash and
signed devices
    if msg.xep0000.hash and self.is_initiator:
        console_string = "Encrypted hash and signed devices"
        print(f'{route}{console_string}')
        asyncio.ensure_future(self.received_hash_from_passive_opponent(msg,
route))
        return msg
```

```
    # if the message is verification request
    if msg.xep0000.verification_start and not self.is_initiator:
        self.received_verification_request(msg, route)
        return msg

    # if the message from the initiator containing the encrypted hash, en-
crypted salt and signed devices
    if msg.xep0000.hash and not self.is_initiator:
        console_string = "Encrypted hash, encrypted salt and signed de-
vices"
        print(f'{route}{console_string}')
        asyncio.ensure_future(self.received_hash_from_initiator(msg,
route))
        return msg

    # if the message contains information about the end of the session
    if msg.xep0000.verification_ended:
        result_text = msg.xep0000.verification_ended.code
        console_string = result_text
        print(f'{route}{console_string}')
        self.finish_verification_session()
        return msg
```

Приложение Б. Скриншоты пользовательского интерфейса

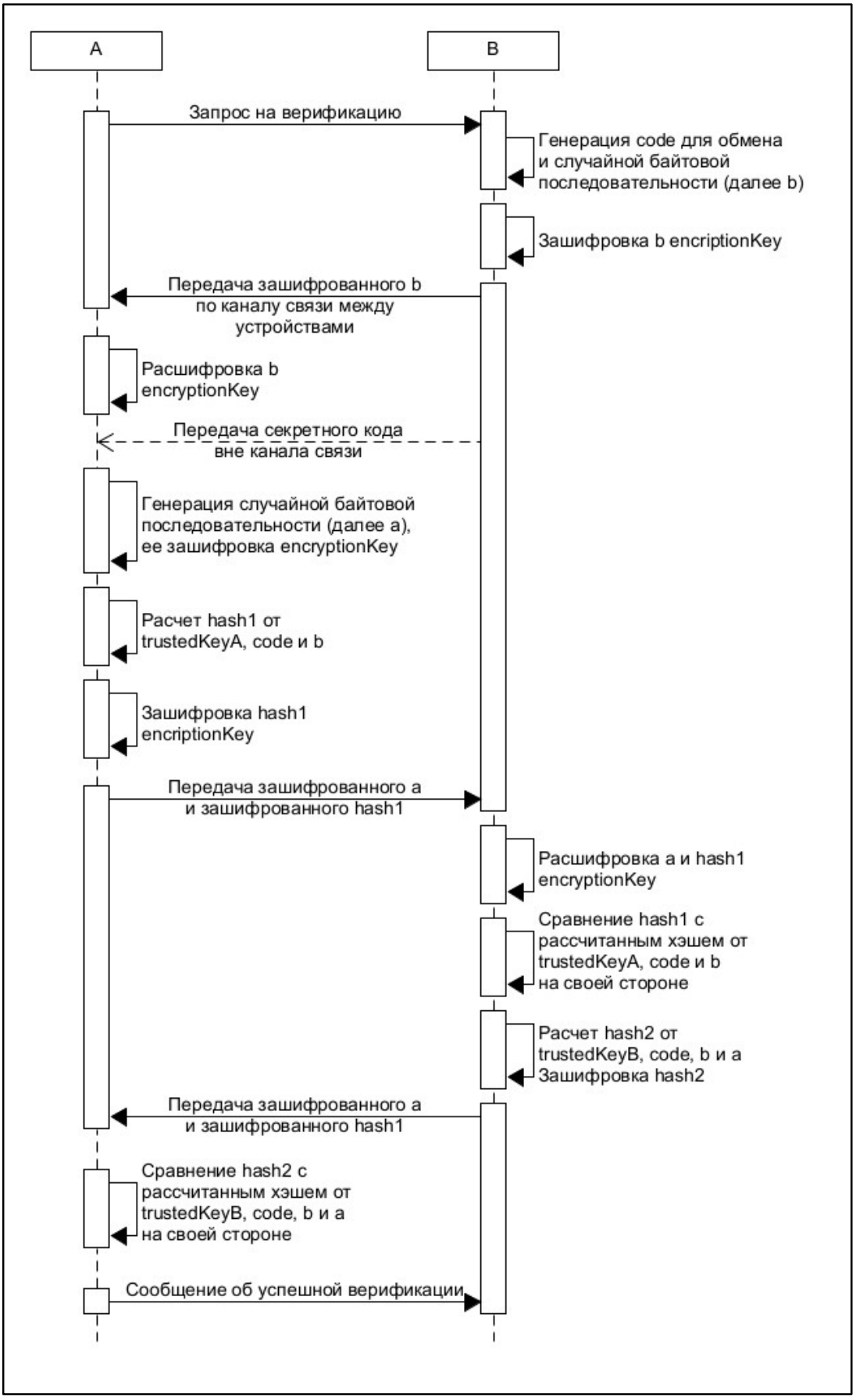

Рисунок 1 – Диаграмма последовательности алгоритма

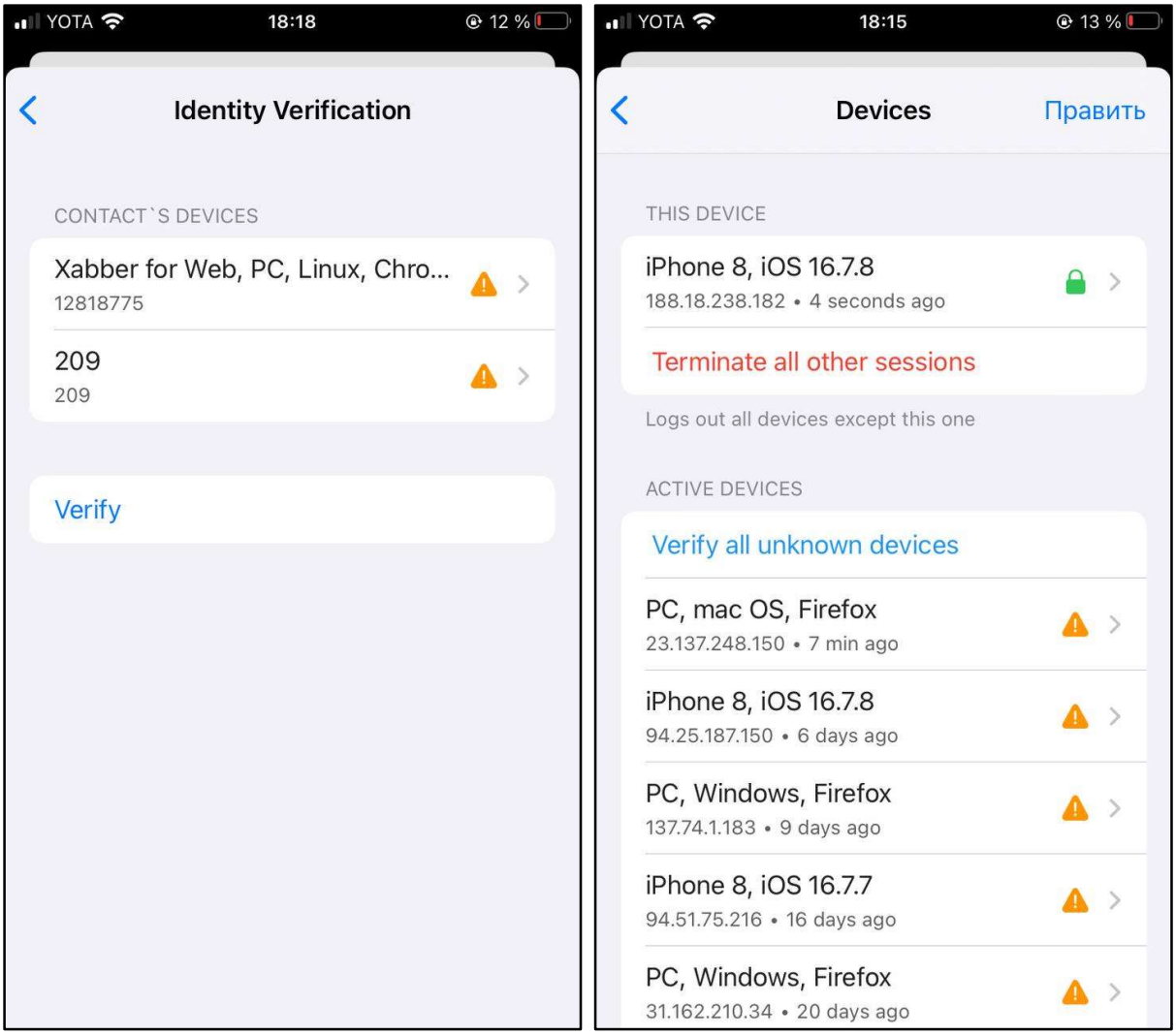

Рисунок 2 – Окно отправления запроса на верификацию пользователю (слева), окно отправления запроса на верификацию своему устройству (справа)

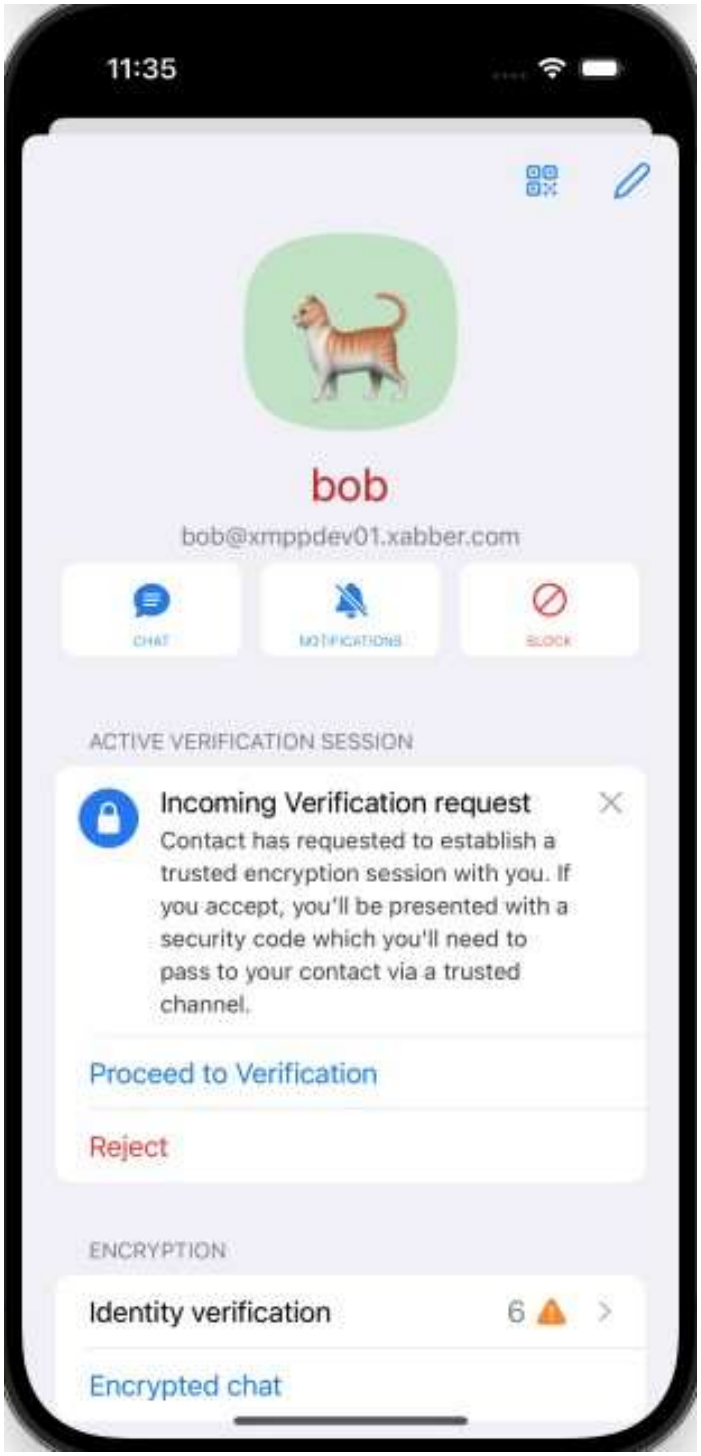

Рисунок 3 – Окно с информацией о сессии верификации

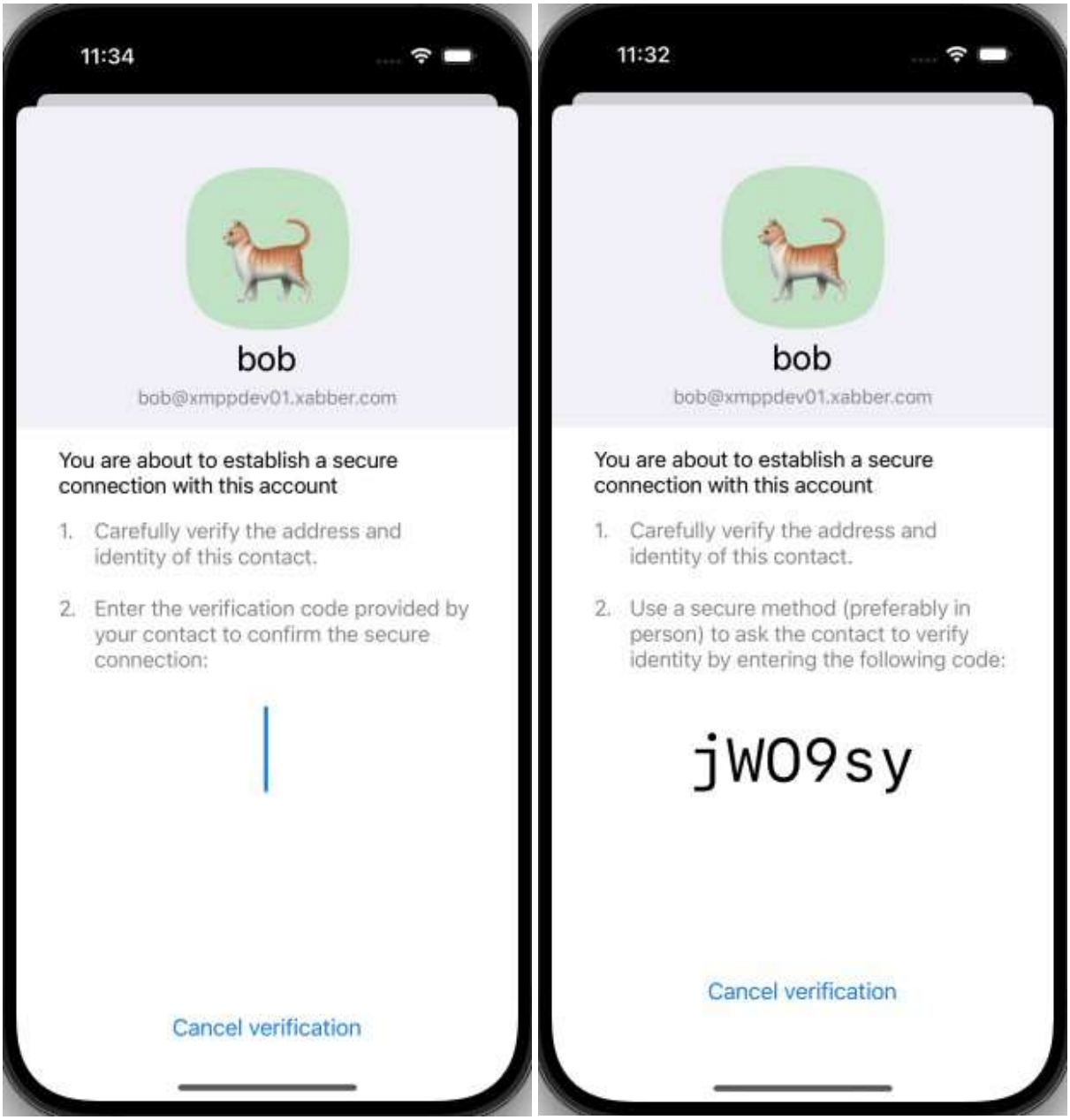

Рисунок 4 – Окна для ввода и для просмотра секретного кода

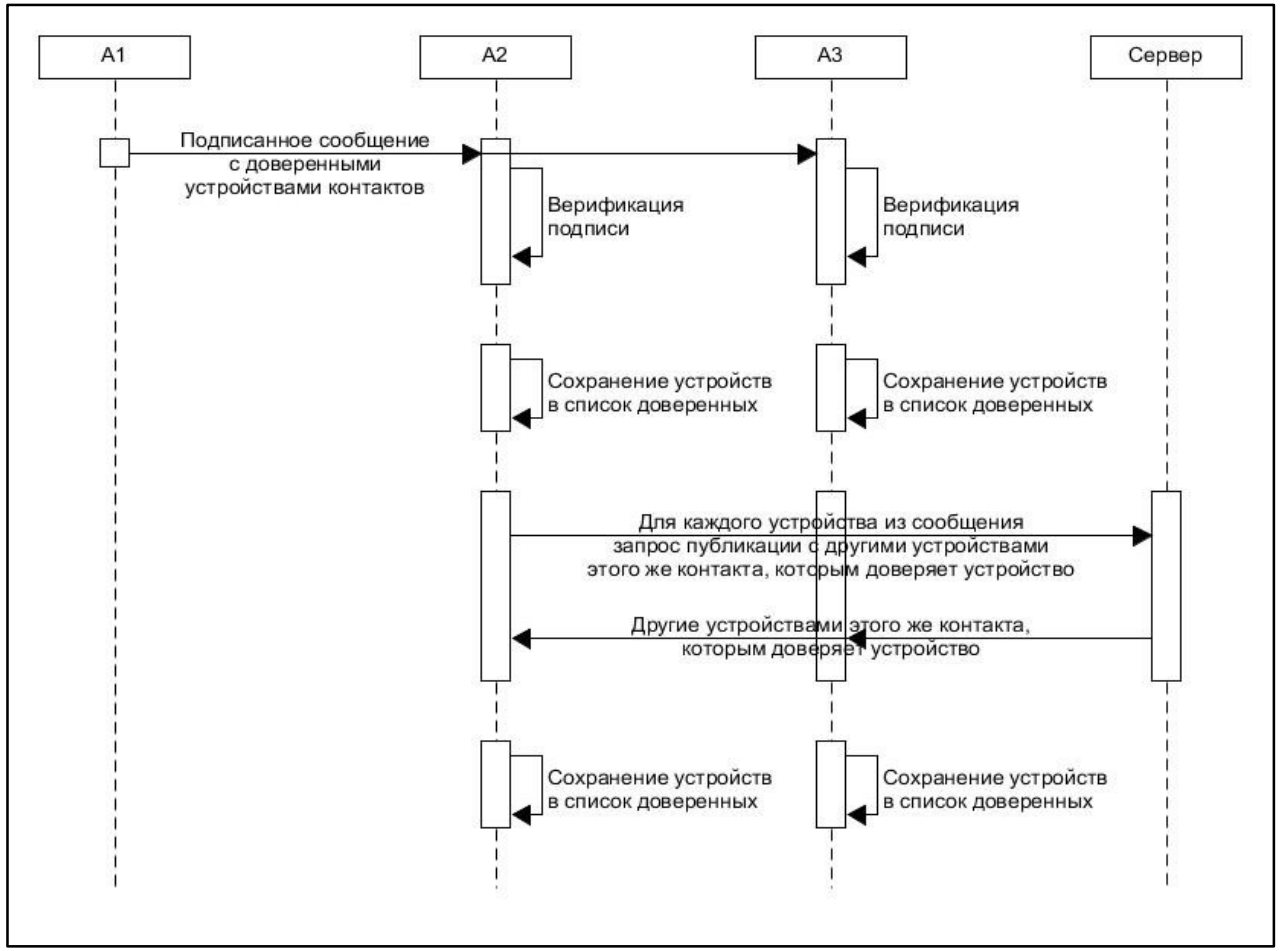

Рисунок 5 – Диаграмма последовательности обмена доверенными устройствами между устройствами одного пользователя

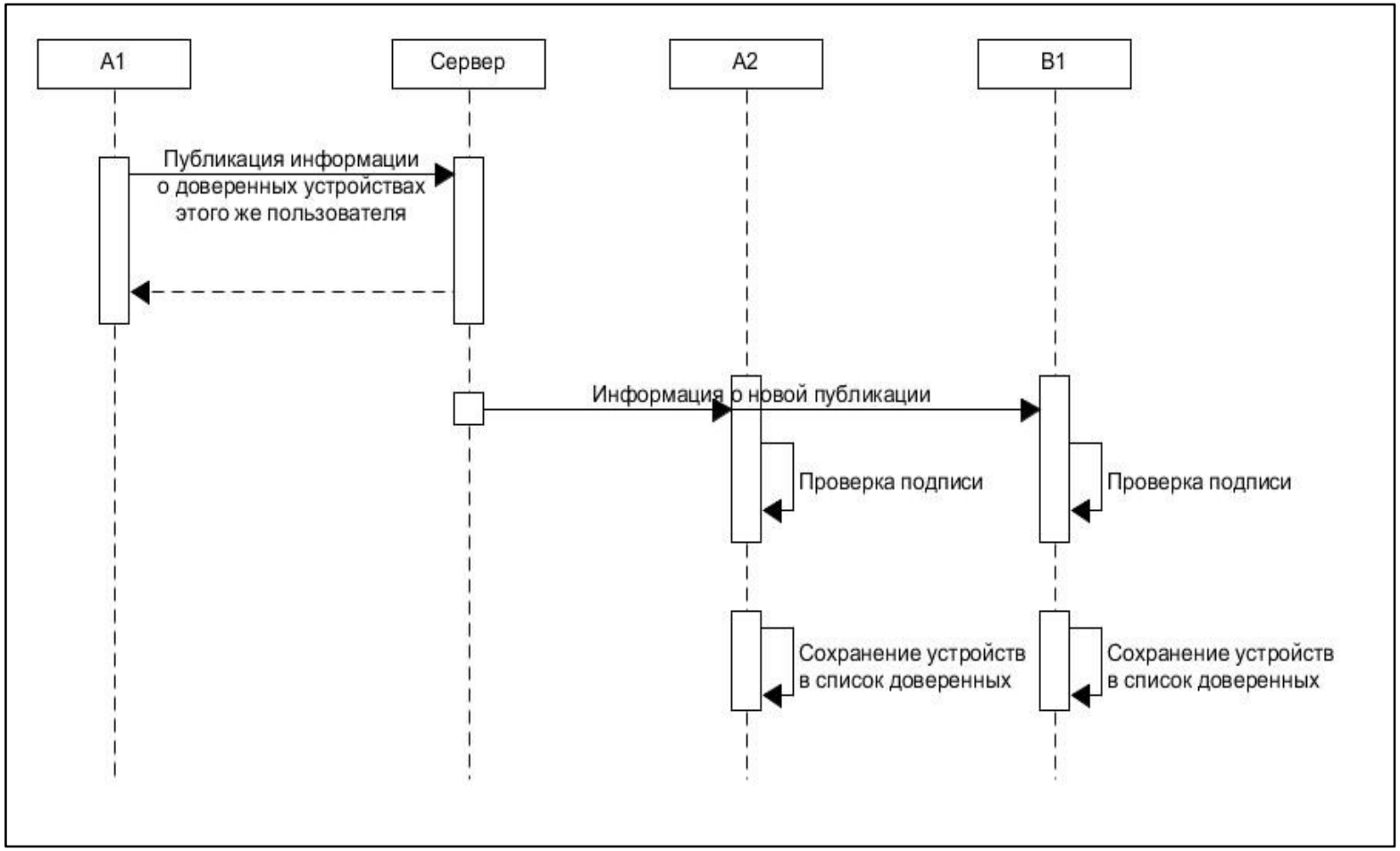

Рисунок 6 – Диаграмма последовательности обмена доверенными устройствами,принадлежащими этому же пользователю