**Федеральное государственное бюджетное образовательное учреждение высшего образования "Белгородский государственный технологический университет им. В.Г. Шухова"**

Кафедра программного обеспечения вычислительной техники и

автоматизированных систем.

**КУРСОВАЯ РАБОТА**

по дисциплине «Криптографические интерфейсы»

на тему:

«Обмен сообщениями с использованием алгоритма Double Ratchet»

Выполнил:

Студент группы КБ-211

Коренев Д.Н.

Принял:

Смакаев А. В.

**Оглавление**

[Введение 3](#_Toc185468626)

[1. Основная часть 5](#_Toc185468627)

[1.1. Постановка задачи 5](#_Toc185468628)

[1.2. Описание решения задачи 6](#_Toc185468629)

[1.3. Ожидаемый результат 6](#_Toc185468630)

[1.4. Техническая информация 6](#_Toc185468631)

[1.4.1. Введение 6](#_Toc185468632)

[1.4.2. Обзор 7](#_Toc185468633)

[1.4.2.1. Цепочки KDF 7](#_Toc185468634)

[1.4.2.2. Храповик с симметричным ключом 9](#_Toc185468635)

[1.4.2.3. Храповик Диффи-Хеллмана 10](#_Toc185468636)

[1.4.2.4. Двойной храповик 17](#_Toc185468637)

[1.4.2.5. Сообщения не по порядку 21](#_Toc185468638)

[1.5. Реализация 23](#_Toc185468639)

[Заключение 26](#_Toc185468640)

[Список использованной литературы 28](#_Toc185468641)

[Приложения 29](#_Toc185468642)

Введение

Современное общество всё больше зависит от цифровых технологий, что делает вопросы информационной безопасности и защиты данных чрезвычайно актуальными. В условиях постоянно возрастающих угроз, таких как кибератаки, перехват сообщений и несанкционированный доступ к конфиденциальной информации, разработка и внедрение надёжных криптографических протоколов становится одной из важнейших задач.

Одним из ключевых элементов безопасной коммуникации является алгоритм Double Ratchet, который используется в таких популярных протоколах, как Signal Protocol, применяемый мессенджерами Signal, WhatsApp и другими. Double Ratchet обеспечивает сквозное шифрование и высокую устойчивость к компрометации ключей благодаря динамической смене ключей шифрования после каждой сессии обмена сообщениями. Это делает алгоритм особенно ценным для обеспечения конфиденциальности и целостности данных.

Целью данной курсовой работы является реализация алгоритма Double Ratchet, анализ его архитектуры и принципов работы, а также оценка эффективности и безопасности. Работа включает изучение теоретических аспектов криптографических протоколов, реализацию алгоритма с использованием современного инструментария программирования и тестирование его работоспособности на примере модельного приложения.

Задачи исследования включают:

1. Изучение принципов работы алгоритма Double Ratchet и его компонентов, включая ратчет ключей и ратчет сообщений.

2. Реализацию алгоритма в программной среде.

3. Анализ безопасности алгоритма в контексте возможных атак.

4. Оценку производительности реализованного решения.

Актуальность данной темы обусловлена необходимостью разработки надёжных методов защиты данных в условиях быстрого развития цифровой среды и растущего количества угроз информационной безопасности. Работа представляет интерес как для теоретического изучения криптографических систем, так и для практического применения в разработке защищённых коммуникационных приложений.

1. Основная часть

1.1. Постановка задачи

В условиях современного развития цифровых технологий обеспечение конфиденциальности и безопасности передачи данных является одной из ключевых проблем. Алгоритм Double Ratchet, используемый в протоколах для сквозного шифрования, позволяет эффективно защищать коммуникации от компрометации даже при частичном компрометировании ключей. Однако реализация этого алгоритма требует глубокого понимания криптографических методов и грамотного программного исполнения, что представляет собой сложную инженерную задачу.

Основной целью данной курсовой работы является разработка и реализация алгоритма Double Ratchet, который будет удовлетворять следующим требованиям:

1. Обеспечение безопасного динамического обновления ключей шифрования при каждом обмене сообщениями.

2. Защита данных от атак на конфиденциальность, таких как перехват сообщений или ретрансляция скомпрометированных данных.

3. Эффективное функционирование в условиях реального времени с минимальными вычислительными затратами.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Изучить архитектуру алгоритма Double Ratchet, включая работу ратчета ключей (Key Ratchet) и ратчета сообщений (Message Ratchet).

2. Разработать программную реализацию алгоритма, используя выбранный язык программирования.

3. Провести тестирование алгоритма на примере моделируемого приложения для обмена сообщениями.

Результатом работы станет функциональная реализация алгоритма Double Ratchet, подтверждённая теоретическим обоснованием и экспериментальным анализом, что позволит глубже понять возможности и ограничения данного алгоритма для защиты данных в современных коммуникационных системах.

1.2. Описание решения задачи

Для решения поставленной задачи была выбрана реализация алгоритма Double Ratchet на языке программирования Rust. Rust был выбран благодаря его безопасности, производительности и встроенной поддержке работы с системным уровнем, что позволяет эффективно управлять памятью и предотвращать типичные уязвимости, такие как переполнение буфера и использование освобождённых указателей. Эти свойства делают Rust идеальным инструментом для разработки криптографических систем.

1.3. Ожидаемый результат

На выходе реализации будет готовое решение, позволяющее безопасно шифровать сообщения в режиме реального времени. Реализация отличается высокой производительностью, надёжностью и устойчивостью к типовым угрозам безопасности.

1.4. Техническая информация

1.4.1. Введение

Алгоритм Double Ratchet используется двумя сторонами для обмена зашифрованными сообщениями на основе общего секретного ключа. Обычно стороны используют какой-либо протокол согласования ключей (например, X3DH (Extended Triple Diffie-Hellman)) для согласования общего секретного ключа. После этого стороны используют алгоритм Double Ratchet для отправки и получения зашифрованных сообщений.

Стороны получают новые ключи для каждого сообщения Double Ratchet, так что более ранние ключи не могут быть вычислены из более поздних. Стороны также отправляют открытые значения Диффи-Хеллмана, прикрепленные к их сообщениям. Результаты вычислений Диффи-Хеллмана подмешиваются к полученным ключам, так что более поздние ключи не могут быть вычислены из более ранних. Эти свойства обеспечивают определенную защиту более ранних или более поздних зашифрованных сообщений в случае компрометации ключей одной из сторон.

Ниже представлены "Двойной храповик" и его вариант шифрования заголовков, а также обсуждаются их защитные свойства.

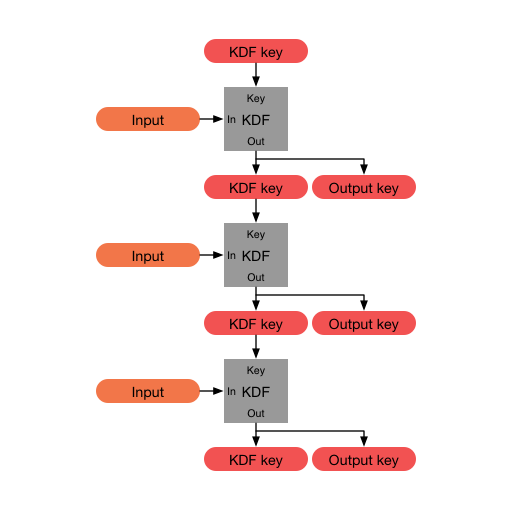
1.4.2. Обзор

1.4.2.1. Цепочки KDF

**Цепочка KDF** (key derivation function - функция формирования ключа)является основным понятием алгоритма Double Ratchet.

Мы определяем **KDF** как криптографическую функцию, которая принимает секретный случайный **ключ KDF** и некоторые входные данные и возвращает выходные данные. Выходные данные неотличимы от случайных при условии, что ключ неизвестен (то есть KDF удовлетворяет требованиям криптографического "PRF" (pseudorandom function family)). Если ключ не является секретным и случайным, KDF все равно должен обеспечивать безопасный криптографический хэш своего ключа и входных данных. Конструкции HMAC и HKDF при использовании безопасного хэш-алгоритма удовлетворяют определению KDF.

Мы используем термин **"цепочка KDF"**, когда часть выходных данных KDF используется в качестве **выходного ключа**, а часть - для замены ключа KDF, который затем может быть использован с другим входом. На следующей диаграмме представлена цепочка KDF, обрабатывающая три входа и производящая три выходных ключа:

  
Рисунок 1. Цепочка KDF.

Цепочка KDF обладает следующими свойствами:

**Устойчивость:** Выходные ключи кажутся случайными для противника, не знающего ключи KDF. Это верно, даже если противник может контролировать входы KDF.

**Защита наперед:** Противнику, узнавшему ключ KDF в определенный момент времени, выходные ключи из прошлого кажутся случайными.

**Восстановление после взлома:** Будущие выходные ключи кажутся случайными для противника, который узнает ключ KDF в какой-то момент времени, при условии, что будущие входные данные добавили достаточную энтропию.

В **сессии Double Ratchet** между Алисой и Бобом каждая сторона хранит ключ KDF для трех цепочек: **корневой**, **отправляющей** и **принимающей** (отправляющая цепочка Алисы совпадает с принимающей цепочкой Боба, и наоборот).

Когда Алиса и Боб обмениваются сообщениями, они также обмениваются новыми открытыми ключами Диффи-Хеллмана, а выходные секреты Диффи-Хеллмана становятся входными для корневой цепочки. Выходные ключи корневой цепочки становятся новыми ключами KDF для цепочек отправителей и получателей. Это называется **храповиком Диффи-Хеллмана**.

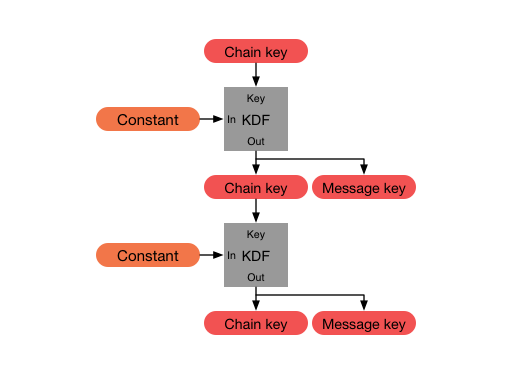
Цепочки отправителей и получателей продвигаются вперед по мере отправки и получения каждого сообщения. Их выходные ключи используются для шифрования и дешифрования сообщений. Это называется **храповиком с симметричным ключом.**

В следующих разделах более подробно объясняются храповики симметричных ключей и Диффи-Хеллмана, а затем показано, как они объединяются в двойной храповик.

1.4.2.2. Храповик с симметричным ключом

Каждое отправленное или полученное сообщение шифруется уникальным **ключом сообщения**. Ключи сообщений являются выходными ключами цепочек KDF отправителя и получателя. Ключи KDF для этих цепочек будут называться **ключами цепочки**.

Входы KDF для цепочек отправки и получения постоянны, поэтому эти цепочки не обеспечивают восстановления после взлома. Цепочки отправки и получения просто обеспечивают шифрование каждого сообщения уникальным ключом, который может быть удален после шифрования или дешифрования. Вычисление следующего ключа цепочки и ключа сообщения из заданного ключа цепочки - это один **шаг храповика** в **храповике симметричных ключей**. На приведенной ниже схеме показаны два шага:

  
Рисунок 2. Храповик с симметричным ключом.

Поскольку ключи сообщений не используются для получения других ключей, ключи сообщений могут храниться без ущерба для безопасности других ключей сообщений. Это полезно для обработки потерянных или неупорядоченных сообщений.

1.4.2.3. Храповик Диффи-Хеллмана

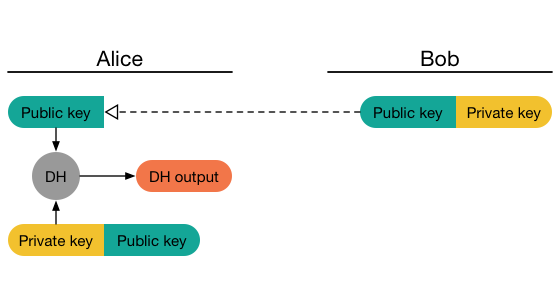
Если злоумышленник украдет цепные ключи одной из сторон, он сможет вычислить все будущие ключи сообщений и расшифровать все будущие сообщения. Чтобы предотвратить это, двойной храповик объединяет храповик с симметричным ключом с **храповиком DH**, который обновляет ключи цепочки на основе результатов Диффи-Хеллмана.

Для реализации храповика DH каждая сторона генерирует пару ключей DH (открытый ключ и закрытый ключ Диффи-Хеллмана), которая становится их текущей **парой ключей храповика**. Каждое сообщение от любой стороны начинается с заголовка, содержащего текущий открытый ключ храповика отправителя. Когда от удаленной стороны поступает новый открытый ключ храповика, выполняется **шаг храповика DH**, который заменяет текущую пару ключей храповика локальной стороны новой парой ключей.

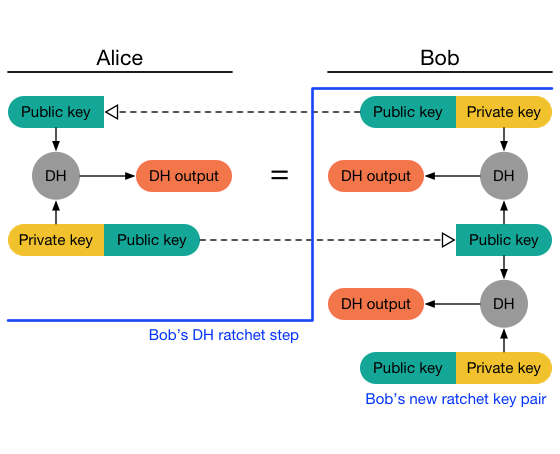
Это приводит к "пинг-понгу", когда стороны по очереди заменяют пары ключей-храповиков. Подслушивающее лицо, ненадолго скомпрометировавшее одну из сторон, может узнать значение текущего закрытого ключа храповика, но этот закрытый ключ в конце концов будет заменен на некомпрометированный. В этот момент вычисления Диффи-Хеллмана между парами ключей-храповиков будут определять выход DH, неизвестный злоумышленнику.

На следующих диаграммах показано, как храповик DH извлекает общую последовательность выходов DH.

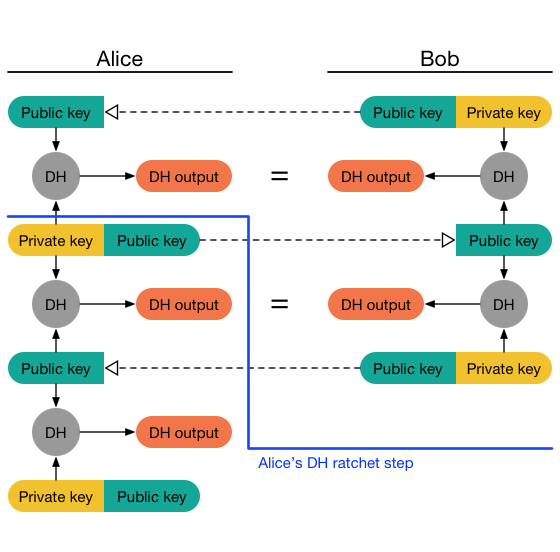
Алиса инициализируется открытым ключом храповика Боба. Открытый ключ храповика Алисы еще не известен Бобу. В процессе инициализации Алиса выполняет вычисление DH между своим закрытым ключом храповика и открытым ключом храповика Боба:

  
Рисунок 3. Храповик Диффи-Хеллмана. Шаг 1.

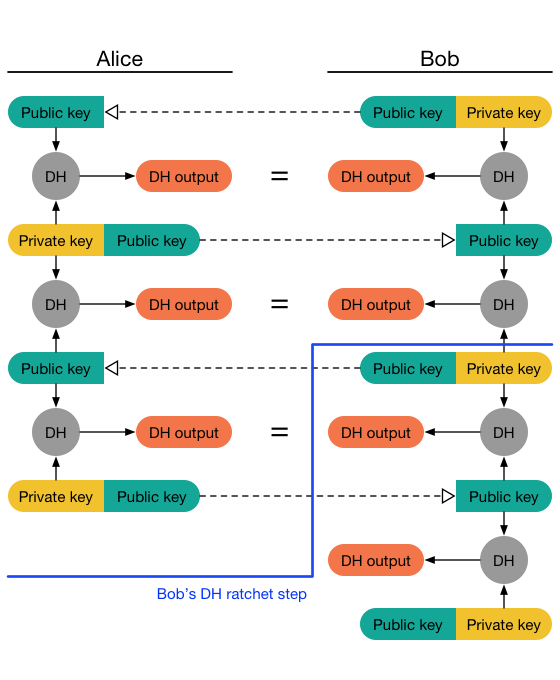
Начальные сообщения Алисы афишируют ее открытый ключ с храповиком. Как только Боб получает одно из этих сообщений, Боб выполняет шаг храповика DH: он вычисляет результат DH между открытым ключом храповика Алисы и своим закрытым ключом храповика, который равен начальному результату DH Алисы. Затем Боб заменяет свою пару ключей храповика и вычисляет новый результат DH:

  
Рисунок 4. Храповик Диффи-Хеллмана. Шаг 2.

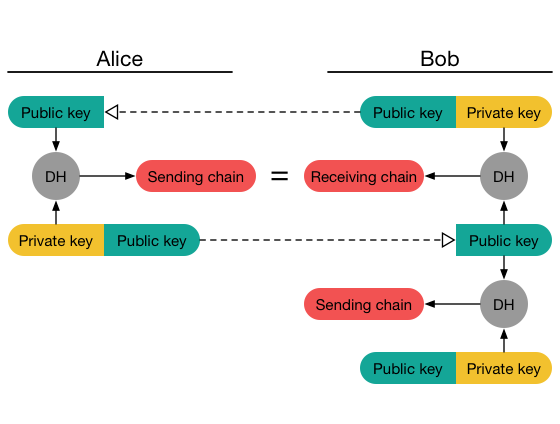
Сообщения, отправленные Бобом, афишируют его новый открытый ключ. В конце концов Алиса получит одно из сообщений Боба и выполнит шаг храповика DH, заменив свою пару ключей храповика и получив два выхода DH: один, соответствующий последнему ключу Боба, и новый:

  
Рисунок 5. Храповик Диффи-Хеллмана. Шаг 3.

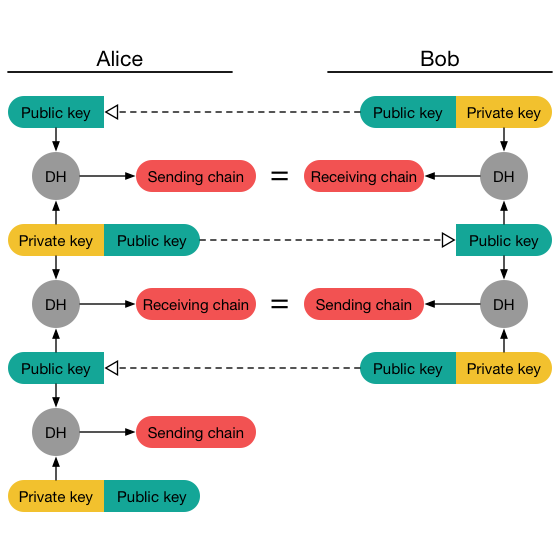
Сообщения, отправленные Алисой, афишируют ее новый открытый ключ. В конце концов Боб получит одно из этих сообщений и выполнит второй шаг храповика DH, и так далее:

  
Рисунок 6. Храповик Диффи-Хеллмана. Шаг 4.

Результаты DH, сгенерированные на каждом шаге храповика DH, используются для получения новых цепных ключей отправки и получения. На приведенной ниже диаграмме рассматривается первый шаг храповика Боба. Боб использует свой первый вывод DH для получения цепочки получения, которая совпадает с цепочкой отправки Алисы. Боб использует второй результат DH для получения новой цепочки отправки:

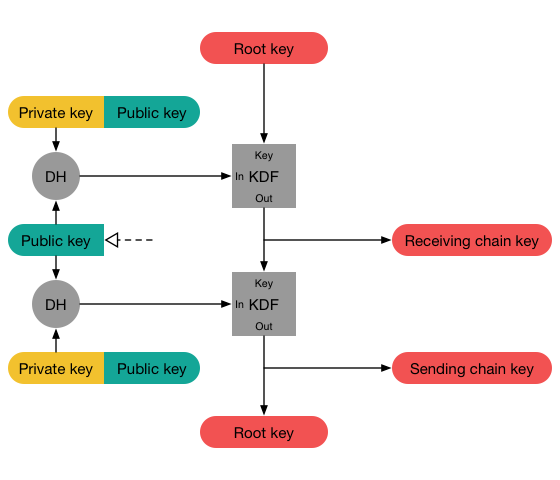
  
Рисунок 7. Храповик Диффи-Хеллмана. Первый шаг Боба.

По мере того, как стороны поочередно выполняют шаги храповика DH, они поочередно вводят новые цепочки отправки:

  
Рисунок 8. Храповик Диффи-Хеллмана. Несколько шагов храповика.

Однако приведенная выше картина является упрощением. Вместо того чтобы получать ключи цепочки непосредственно из выходов DH, выходы DH используются в качестве входов KDF в корневую цепочку, а выходы KDF из корневой цепочки используются в качестве ключей цепочки для отправки и получения. Использование цепочки KDF повышает устойчивость к взлому и восстанавливает работоспособность.

Таким образом, полный шаг храповика DH состоит из двойного обновления корневой цепочки KDF и использования выходных ключей KDF в качестве новых ключей цепочки приема и отправки:

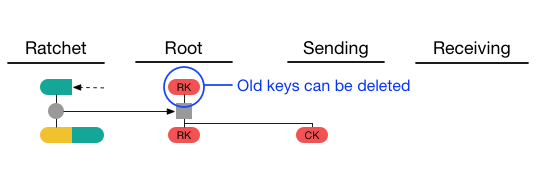
  
Рисунок 9. Храповик Диффи-Хеллмана. Обобщенная схема.

1.4.2.4. Двойной храповик

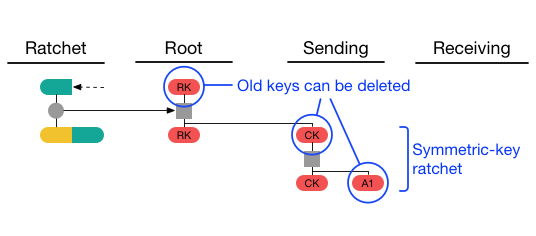
Сочетание симметрично-ключевого и DH-храповика дает Двойной храповик:

* Когда сообщение отправляется или принимается, к цепочке отправителей или получателей применяется шаг храповика симметричного ключа для получения ключа сообщения.
* Когда получен новый открытый ключ храповика, перед храповиком симметричных ключей выполняется шаг храповика DH для замены цепных ключей.

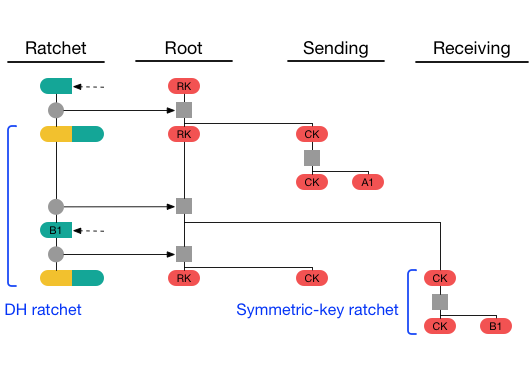
На приведенной ниже схеме Алиса была инициализирована с открытым ключом храповика Боба и общим секретом, который является начальным корневым ключом. В процессе инициализации Алиса генерирует новую пару храповых ключей и передает результат DH в корневой KDF для вычисления нового корневого ключа (*RK*) и ключа цепочки отправки (*CK*):

  
Рисунок 10. Двойной храповик. Инициализация Алисы.

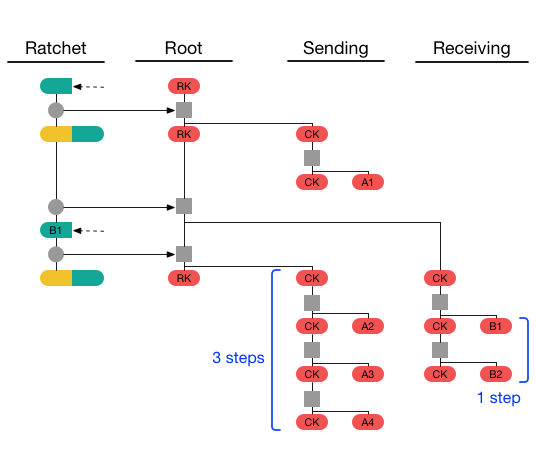
Когда Алиса отправляет свое первое сообщение *A1*, она применяет шаг храповика симметричного ключа к своему цепному ключу отправки, в результате чего получается новый ключ сообщения (ключи сообщений будут помечены сообщением, которое они шифруют или расшифровывают). Новый цепной ключ сохраняется, а ключ сообщения и старый цепной ключ могут быть удалены:

  
Рисунок 11. Двойной храповик. Первое сообщение Алисы.

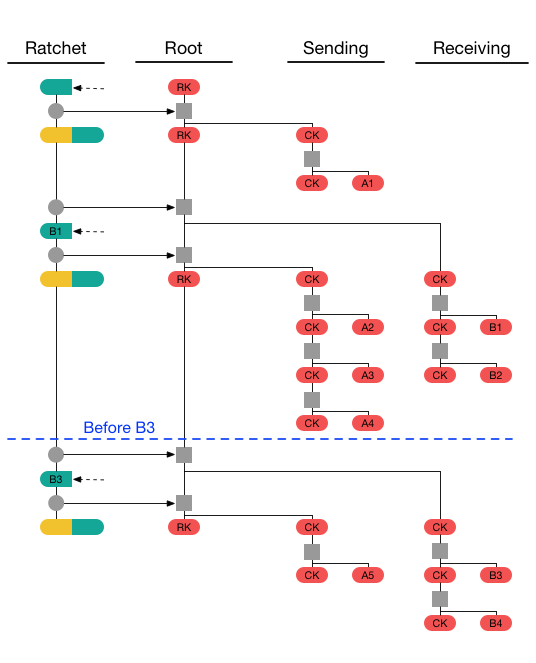
Если Алиса в следующий раз получит ответ *B1* от Боба, он будет содержать новый открытый ключ с храповиком (открытые ключи Боба помечены сообщением, когда они были впервые получены). Алиса применяет шаг храповика DH для получения новых цепных ключей получения и отправки. Затем она применяет шаг храповика с симметричным ключом к цепочке получения, чтобы получить ключ сообщения для полученного сообщения:

  
Рисунок 12. Двойной храповик. Шаг храповика DH для получения новых цепных ключей получения и отправки.

Предположим, что Алиса отправляет сообщение *A2*, получает сообщение *B2* со старым открытым ключом Боба с храповиком, затем отправляет сообщения *A3* и *A4*. Цепочка отправки Алисы будет храниться три шага, а цепочка получения - один:

  
Рисунок 13. Двойной храповик. Хранение цепочек.

Предположим, что Алиса получает сообщения *B3* и *B4* со следующим ключом храповика Боба, а затем отправляет сообщение *A5*. Окончательное состояние Алисы будет следующим:

  
Рисунок 14. Двойной храповик. Окончательное состояние.

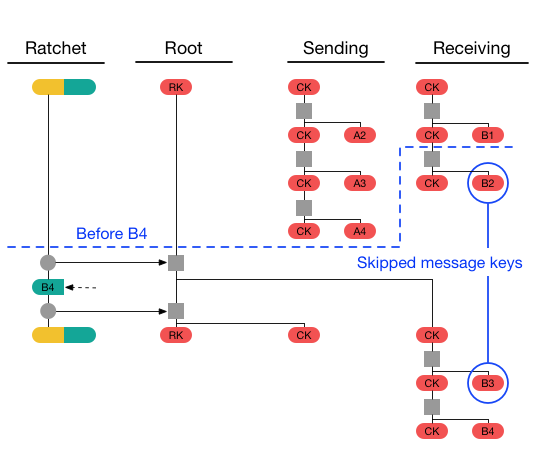
1.4.2.5. Сообщения не по порядку

Double Ratchet обрабатывает потерянные или не по порядку сообщения, включая в заголовок каждого сообщения номер сообщения в цепочке отправки (*N=0*,1,2,...) и длину (количество ключей сообщений) в предыдущей цепочке отправки (*PN*). Это позволяет получателю перейти к соответствующему ключу сообщения, сохраняя пропущенные ключи на случай, если пропущенные сообщения придут позже.

При получении сообщения, если срабатывает шаг храповика DH, то полученное *значение PN* минус длина текущей цепочки приема - это количество пропущенных сообщений в этой цепочке приема. Полученное *N -* это количество пропущенных сообщений в новой цепочке приема (т. е. в цепочке после храповика DH).

Если шаг храповика DH не срабатывает, то полученное значение *N* минус длина принимающей цепочки - это количество пропущенных сообщений в этой цепочке.

Например, рассмотрим последовательность сообщений из предыдущего раздела, когда сообщения *B2* и *B3* пропущены. Сообщение *B4* запустит храповой шаг DH Алисы (вместо *B3*). Сообщение *B4* будет иметь *PN=2* и *N=1*. При получении *B4* у Алисы будет цепочка приема длины 1 (*B1*), поэтому Алиса будет хранить ключи сообщений *B2* и *B3*, чтобы их можно было расшифровать, если они придут позже:

  
Рисунок 15. Двойной храповик. Пропущенные сообщения.

1.5. Реализация

Полный код реализации представлен в приложениях 1–3. На изображениях ниже показана работа разработанного приложения.

На рисунке 16 выполнена инициализация сервера для обмена сообщениями.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание  
Рисунок 16. Инициализация сервера.

После инициализации сервера мы можем начать подключение клиентов. На рисунке 17 показано подключение клиента с идентификатором “a” с запросом на подключение к клиенту “b”.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание  
Рисунок 17. Инициализация первого клиента.

Пока ответный клиент не инициализирован, мы не можем отправлять сообщения (рисунок 18).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание  
Рисунок 18. Без второго активного клиента обмен сообщениями невозможен.

Как только клиент-ответчик подключается к серверу, происходит обмен ключами и инициализируется сессия (рисунок 19).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описаниеРисунок 19. Как только второй клиент инициализирован, сессия создана.

Теперь мы можем пересылать сообщения между клиентами и быть уверенными что наши данные зашифрованы (рисунок 20).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание  
Рисунок 20. Обмен сообщениями.

На рисунке 21 показан внутренний лог сервера во время обмена сообщениями.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание  
Рисунок 21. Вид со стороны сервера.

Заключение

В ходе выполнения курсовой работы была успешно реализована программная версия алгоритма Double Ratchet на языке программирования Rust. Реализация включает полный цикл работы алгоритма: от инициализации сеанса и генерации ключей до шифрования и дешифрования сообщений с использованием механизмов динамического обновления ключей.

Результаты работы продемонстрировали, что алгоритм Double Ratchet способен обеспечивать высокую степень безопасности и конфиденциальности данных в условиях реального времени. Проведённые тесты подтвердили соответствие реализации заявленным требованиям:

- Прямая и обратная секретность: Каждый ключ используется только один раз, а компрометация ключа не ставит под угрозу предыдущие или последующие сообщения.

- Устойчивость к атакам: Алгоритм успешно противостоит атакам перехвата, ретрансляции и компрометации.

- Производительность: Реализация показала низкие вычислительные затраты, что позволяет использовать алгоритм даже на устройствах с ограниченными ресурсами.

Работа доказала эффективность выбранного подхода к решению задачи, а также продемонстрировала преимущества использования языка Rust для разработки криптографических систем. Безопасность работы с памятью и высокий уровень контроля над процессами сделали его оптимальным выбором для реализации криптографических алгоритмов.

Разработанная программа может быть использована как базовый компонент для создания защищённых систем обмена сообщениями, таких как мессенджеры или IoT-устройства. Работа имеет значительный потенциал для дальнейших исследований:

1. Расширение функциональности: Добавление поддержки групповых чатов или асинхронного обмена данными.

2. Интеграция с существующими системами: Внедрение Double Ratchet в готовые протоколы, такие как Signal Protocol.

3. Оптимизация производительности: Улучшение скорости работы при большом объёме сообщений.

Таким образом, данная работа вносит вклад в развитие современных методов защиты данных и подтверждает актуальность использования алгоритма Double Ratchet в условиях растущих требований к информационной безопасности.

Список использованной литературы

1. Васильева, И. Н. Криптографические методы защиты информации : учебник и практикум для вузов / И. Н. Васильева. — Москва : Издательство Юрайт, 2024. — 310 с.
2. Фомичёв, В. М. Криптографические методы защиты информации в 2 ч. Часть 2. Системные и прикладные аспекты : учебник для вузов / В. М. Фомичёв, Д. А. Мельников ; под редакцией В. М. Фомичёва. — Москва : Издательство Юрайт, 2024. — 245 с.
3. Запечников, С. В. Криптографические методы защиты информации : учебник для вузов / С. В. Запечников, О. В. Казарин, А. А. Тарасов. — Москва : Издательство Юрайт, 2024. — 309 с.
4. Бабичев, С. Л. Распределенные системы : учебное пособие для вузов / С. Л. Бабичев, К. А. Коньков. — Москва : Издательство Юрайт, 2024. — 507 с.
5. Трофимов, В. В. Глобальные и локальные сети : учебник для вузов / В. В. Трофимов, М. И. Барабанова, В. И. Кияев. — 4-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2024. — 151 с.
6. Jo¨el Alwen. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol / Jo¨el Alwen, Sandro Coretti, Yevgeniy Dodis — New York : New York University, 2020. — 56 с.
7. Signal Double Ratchet Protocol // Signal : [сайт] – URL: https://signal.org/docs/specifications/doubleratchet/ (дата обращения 29.11.2024).
8. Advanced cryptographic ratcheting // Signal : [сайт] – URL: https://signal.org/blog/advanced-ratcheting/ (дата обращения 27.11.2024).

Приложения

Приложение 1. Реализация алгоритма Double Ratchet.

use core::{cmp, fmt, hash::Hash};

use hashbrown::HashMap;

use rand\_core::{CryptoRng, RngCore};

#[cfg(feature = "std")]

use std::error::Error;

use alloc::vec::Vec;

// Upper limit on the receive chain ratchet steps when trying to decrypt. Prevents a

// denial-of-service attack where the attacker

const MAX\_SKIP: usize = 1000;

/// Message Counter (as seen in the header)

pub type Counter = u32;

pub struct DoubleRatchet<CP: CryptoProvider> {

    dhs: CP::KeyPair,

    dhr: Option<CP::PublicKey>,

    rk: CP::RootKey,

    cks: Option<CP::ChainKey>,

    ckr: Option<CP::ChainKey>,

    ns: Counter,

    nr: Counter,

    pn: Counter,

    mkskipped: KeyStore<CP>,

}

impl<CP> fmt::Debug for DoubleRatchet<CP>

where

    CP: CryptoProvider,

    CP::KeyPair: fmt::Debug,

    CP::PublicKey: fmt::Debug,

    CP::RootKey: fmt::Debug,

    CP::ChainKey: fmt::Debug,

    CP::MessageKey: fmt::Debug,

{

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(

            f,

            "DoubleRatchet {{ dhs: {:?}, dhr: {:?}, rk: {:?}, cks: {:?}, ckr: {:?}, ns: {:?}, \

             nr: {:?}, pn: {:?}, mkskipped: {:?} }}",

            self.dhs,

            self.dhr,

            self.rk,

            self.cks,

            self.ckr,

            self.ns,

            self.nr,

            self.pn,

            self.mkskipped

        )

    }

}

impl<CP: CryptoProvider> DoubleRatchet<CP> where {

    /// Initialize "Alice": the sender of the first message.

    ///

    /// This implements `RatchetInitAlice` as defined in the [specification] when `initial\_receive

    /// = None`: after initialization Alice must send a message to Bob before he is able to provide

    /// a reply.

    ///

    /// Alternatively Alice provides an extra symmetric key: `initial\_receive = Some(key)`, so that

    /// both Alice and Bob can send the first message. Note however that even when Alice and Bob

    /// initialize this way the initialization is asymmetric in the sense that Alice requires Bob's

    /// public key.

    ///

    /// Either Alice and Bob must supply the same extra symmetric key or both must supply `None`.

    ///

    /// # Security considerations

    ///

    /// For security, initialization through `new\_alice` has the following requirements:

    ///  - `shared\_secret` must be both \*confidential\* and \*authenticated\*

    ///  - `them` must be \*authenticated\*

    ///  - `initial\_receive` is `None` or `Some(key)` where `key` is \*confidential\* and \*authenticated\*

    ///

    /// [specification]: https://signal.org/docs/specifications/doubleratchet/#initialization

    pub fn new\_alice<R: CryptoRng + RngCore>(

        shared\_secret: &CP::RootKey,

        them: CP::PublicKey,

        initial\_receive: Option<CP::ChainKey>,

        rng: &mut R,

    ) -> Self {

        let dhs = CP::KeyPair::new(rng);

        let (rk, cks) = CP::kdf\_rk(shared\_secret, &CP::diffie\_hellman(&dhs, &them));

        Self {

            dhs,

            dhr: Some(them),

            rk,

            cks: Some(cks),

            ckr: initial\_receive,

            ns: 0,

            nr: 0,

            pn: 0,

            mkskipped: KeyStore::new(),

        }

    }

    /// Initialize "Bob": the receiver of the first message.

    ///

    /// This implements `RatchetInitBob` as defined in the [specification] when `initial\_send =

    /// None`: after initialization Bob must receive a message from Alice before he can send his

    /// first message.

    ///

    /// Alternatively Bob provides an extra symmetric key: `initial\_send = Some(key)`, so that both

    /// Alice and Bob can send the first message. Note however that even when Alice and Bob

    /// initialize this way the initialization is asymmetric in the sense that Bob must provide his

    /// public key to Alice.

    ///

    /// Either Alice and Bob must supply the same extra symmetric key or both must supply `None`.

    ///

    /// # Security considerations

    ///

    /// For security, initialization through `new\_bob` has the following requirements:

    ///  - `shared\_secret` must be both \*confidential\* and \*authenticated\*

    ///  - the private key of `us` must remain secret on Bob's device

    ///  - `initial\_send` is `None` or `Some(key)` where `key` is \*confidential\* and \*authenticated\*

    ///

    /// [specification]: https://signal.org/docs/specifications/doubleratchet/#initialization

    pub fn new\_bob(

        shared\_secret: CP::RootKey,

        us: CP::KeyPair,

        initial\_send: Option<CP::ChainKey>,

    ) -> Self {

        Self {

            dhs: us,

            dhr: None,

            rk: shared\_secret,

            cks: initial\_send,

            ckr: None,

            ns: 0,

            nr: 0,

            pn: 0,

            mkskipped: KeyStore::new(),

        }

    }

    /// Try to encrypt the `plaintext`. See `ratchet\_encrypt` for details.

    ///

    /// Fails with `EncryptUninit` when `self` is not yet initialized for encrypting.

    pub fn try\_ratchet\_encrypt<R: CryptoRng + RngCore>(

        &mut self,

        plaintext: &[u8],

        associated\_data: &[u8],

        rng: &mut R,

    ) -> Result<(Header<CP::PublicKey>, Vec<u8>), EncryptUninit> {

        if self.can\_encrypt() {

            Ok(self.ratchet\_encrypt(plaintext, associated\_data, rng))

        } else {

            Err(EncryptUninit)

        }

    }

    /// Encrypt the `plaintext`, ratchet forward and return the (header, ciphertext) pair.

    ///

    /// Implements `RatchetEncrypt` as defined in the [specification]. The header should be sent

    /// along the ciphertext in order for the recipient to be able to `ratchet\_decrypt`. The

    /// ciphertext is encrypted in some

    /// [AEAD](https://en.wikipedia.org/wiki/Authenticated\_encryption) mode, which encrypts the

    /// `plaintext` and authenticates the `plaintext`, `associated\_data` and the header.

    ///

    /// The internal state of the `DoubleRatchet` is automatically updated so that the next message

    /// key be sent with a fresh key.

    ///

    /// Note that `rng` is only used for updating the internal state and not for encrypting the

    /// data.

    ///

    /// # Panics

    ///

    /// Panics if `self` is not initialized for sending yet. If this is a concern, use

    /// `try\_ratchet\_encrypt` instead to avoid panics.

    ///

    /// [specification]: https://signal.org/docs/specifications/doubleratchet/#encrypting-messages

    pub fn ratchet\_encrypt<R: CryptoRng + RngCore>(

        &mut self,

        plaintext: &[u8],

        associated\_data: &[u8],

        rng: &mut R,

    ) -> (Header<CP::PublicKey>, Vec<u8>) {

        // TODO: [sebastianv89] is this the correct place for clear\_stack\_on\_return?

        let (h, mk) = self.ratchet\_send\_chain(rng);

        let pt = CP::encrypt(&mk, plaintext, &Self::concat(&h, associated\_data));

        (h, pt)

    }

    // Are we initialized such that we can encrypt messages?

    fn can\_encrypt(&self) -> bool {

        self.cks.is\_some() || self.dhr.is\_some()

    }

    // Ratcheting forward the DH chain for sending is delayed until the first message in that chain

    // is going to be sent.

    //

    // [specification]: https://signal.org/docs/specifications/doubleratchet/#deferring-new-ratchet-key-generation

    //

    // # Panics

    //

    // Panics if encrypting is not yet initialized

    fn ratchet\_send\_chain<R: CryptoRng + RngCore>(

        &mut self,

        rng: &mut R,

    ) -> (Header<CP::PublicKey>, CP::MessageKey) {

        if self.cks.is\_none() {

            let dhr = self

                .dhr

                .as\_ref()

                .expect("not yet initialized for encryption");

            self.dhs = CP::KeyPair::new(rng);

            let (rk, cks) = CP::kdf\_rk(&self.rk, &CP::diffie\_hellman(&self.dhs, dhr));

            self.rk = rk;

            self.cks = Some(cks);

            self.pn = self.ns;

            self.ns = 0;

        }

        let h = Header {

            dh: self.dhs.public().clone(),

            n: self.ns,

            pn: self.pn,

        };

        let (cks, mk) = CP::kdf\_ck(self.cks.as\_ref().unwrap());

        self.cks = Some(cks);

        self.ns += 1;

        (h, mk)

    }

    /// Verify-decrypt the `ciphertext`, update `self` and return the plaintext.

    ///

    /// Implements `RatchetDecrypt` as defined in the [specification]. Decryption of the ciphertext

    /// includes verifying the authenticity of the `header`, `ciphertext` and `associated\_data`

    /// (optional).

    ///

    /// `self` is automatically updated upon successful decryption. This includes ratcheting

    /// forward the receiving key-chain and DH key-chain (if necessary) and storing the

    /// `MessageKeys` of any skipped messages so these messages can be decrypted if they arrive out

    /// of order.

    ///

    /// Returns a `DecryptError` when the plaintext could not be decrypted: `self` remains

    /// unchanged in that case. There could be many reasons: inspect the returned error-value for

    /// further details.

    ///

    /// [specification]: https://signal.org/docs/specifications/doubleratchet/#decrypting-messages-1

    pub fn ratchet\_decrypt(

        &mut self,

        header: &Header<CP::PublicKey>,

        ciphertext: &[u8],

        associated\_data: &[u8],

    ) -> Result<Vec<u8>, DecryptError> {

        // TODO: [sebastianv89] is this the correct place for clear\_stack\_on\_return?

        let (diff, pt) =

            self.try\_decrypt(header, ciphertext, &Self::concat(&header, associated\_data))?;

        self.update(diff, header);

        Ok(pt)

    }

    // The actual decryption. Gets a (non-mutable) reference to self to ensure that the state is

    // not changed. Upon successful decryption the state must be updated. The minimum amount of work

    // is done in order to retrieve the correct `MessageKey`: the returned `Diff` object contains

    // the result of that work to avoid doing the work again.

    fn try\_decrypt(

        &self,

        h: &Header<CP::PublicKey>,

        ct: &[u8],

        ad: &[u8],

    ) -> Result<(Diff<CP>, Vec<u8>), DecryptError> {

        use Diff::\*;

        if let Some(mk) = self.mkskipped.get(&h.dh, h.n) {

            Ok((OldKey, CP::decrypt(mk, ct, ad)?))

        } else if self.dhr.as\_ref() == Some(&h.dh) {

            let (ckr, mut mks) =

                Self::skip\_message\_keys(self.ckr.as\_ref().unwrap(), self.get\_current\_skip(h)?);

            let mk = mks.pop().unwrap();

            Ok((CurrentChain(ckr, mks), CP::decrypt(&mk, ct, ad)?))

        } else {

            let (rk, ckr) = CP::kdf\_rk(&self.rk, &CP::diffie\_hellman(&self.dhs, &h.dh));

            let (ckr, mut mks) = Self::skip\_message\_keys(&ckr, self.get\_next\_skip(h)?);

            let mk = mks.pop().unwrap();

            Ok((NextChain(rk, ckr, mks), CP::decrypt(&mk, ct, ad)?))

        }

    }

    // Calculate how many messages should be skipped in the current receive chain to get the

    // required `MessageKey`. Also check if `h` is valid.

    fn get\_current\_skip(&self, h: &Header<CP::PublicKey>) -> Result<usize, DecryptError> {

        let skip =

            h.n.checked\_sub(self.nr)

                .ok\_or(DecryptError::MessageKeyNotFound)? as usize;

        if MAX\_SKIP < skip {

            Err(DecryptError::SkipTooLarge)

        } else if self.mkskipped.can\_store(skip) {

            Ok(skip)

        } else {

            Err(DecryptError::StorageFull)

        }

    }

    // Calculate how many messages should be skipped in the next receive chain to get the required

    // `MessageKey`. Also check if `h` is valid.

    fn get\_next\_skip(&self, h: &Header<CP::PublicKey>) -> Result<usize, DecryptError> {

        // without malicious participants this error can only be triggered if the local MessageKey

        // has already been deleted.

        let prev\_skip =

            h.pn.checked\_sub(self.nr)

                .ok\_or(DecryptError::MessageKeyNotFound)? as usize;

        let skip = h.n as usize;

        if MAX\_SKIP < cmp::max(prev\_skip, skip) {

            Err(DecryptError::SkipTooLarge)

        } else if self

            .mkskipped

            .can\_store((prev\_skip + skip).saturating\_sub(1))

        {

            Ok(skip)

        } else {

            Err(DecryptError::StorageFull)

        }

    }

    // Update the internal state. Assumes that the validity of `h` has already been checked.

    fn update(&mut self, diff: Diff<CP>, h: &Header<CP::PublicKey>) {

        use Diff::\*;

        match diff {

            OldKey => self.mkskipped.remove(&h.dh, h.n),

            CurrentChain(ckr, mks) => {

                self.mkskipped.extend(&h.dh, self.nr, mks);

                self.ckr = Some(ckr);

                self.nr = h.n + 1;

            }

            NextChain(rk, ckr, mks) => {

                if self.ckr.is\_some() && self.nr < h.pn {

                    let ckr = self.ckr.as\_ref().unwrap();

                    let (\_, prev\_mks) = Self::skip\_message\_keys(ckr, (h.pn - self.nr - 1) as usize);

                    let dhr = self.dhr.as\_ref().unwrap();

                    self.mkskipped.extend(dhr, self.nr, prev\_mks);

                }

                self.dhr = Some(h.dh.clone());

                self.rk = rk;

                self.cks = None;

                self.ckr = Some(ckr);

                self.nr = h.n + 1;

                self.mkskipped.extend(&h.dh, 0, mks);

            }

        }

    }

    // Do `skip + 1` ratchet steps in the receive chain. Return the last ChainKey

    // and all computed MessageKeys.

    fn skip\_message\_keys(ckr: &CP::ChainKey, skip: usize) -> (CP::ChainKey, Vec<CP::MessageKey>) {

        // Note: should use std::iter::unfold (currently still in nightly)

        let mut mks = Vec::with\_capacity(skip + 1);

        let (mut ckr, mk) = CP::kdf\_ck(&ckr);

        mks.push(mk);

        for \_ in 0..skip {

            let cm = CP::kdf\_ck(&ckr);

            ckr = cm.0;

            mks.push(cm.1);

        }

        (ckr, mks)

    }

    // Concatenate `h` and `ad` in a single byte-vector.

    fn concat(h: &Header<CP::PublicKey>, ad: &[u8]) -> Vec<u8> {

        let mut v = Vec::new();

        v.extend\_from\_slice(ad);

        h.extend\_bytes\_into(&mut v);

        v

    }

}

/// The Header that should be sent alongside the ciphertext.

///

/// The Header contains the information for the `DoubleRatchet` to find the correct `MessageKey` to

/// decrypt the message. It is generated by `ratchet\_encrypt`

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct Header<PublicKey> {

    /// The public half of the key-pair of the sender

    pub dh: PublicKey,

    /// Counts the number of messages that have been sent in the current symmetric ratchet

    pub n: Counter,

    /// Counts the number of messages that have been sent in the previous symmetric ratchet

    pub pn: Counter,

}

impl<PK: AsRef<[u8]>> Header<PK> {

    // yikes

    fn extend\_bytes\_into(&self, v: &mut Vec<u8>) {

        v.extend\_from\_slice(self.dh.as\_ref());

        v.extend\_from\_slice(&self.n.to\_be\_bytes());

        v.extend\_from\_slice(&self.pn.to\_be\_bytes());

    }

}

/// Provider of the required cryptographic types and functions.

///

/// The implementer of this trait provides the `DoubleRatchet` with the required external functions

/// as given in the [specification].

///

/// # Security considerations

///

/// The details of the `CryptoProvider` are critical for providing security of the communication.

/// The `DoubleRatchet` can only guarantee security of communication when instantiated with a

/// `CryptoProvider` with secure types and functions. The [specification] provides some sensible

/// [recommendations] and for example code using the `DoubleRatchet` see `tests/signal.rs`.

///

/// [specification]: https://signal.org/docs/specifications/doubleratchet/#external-functions

/// [recommendations]: https://signal.org/docs/specifications/doubleratchet/#recommended-cryptographic-algorithms

pub trait CryptoProvider {

    /// A public key for use in the Diffie-Hellman calculation.

    ///

    /// It is assumed that a `PublicKey` holds a valid key, so if any verification is required the

    /// constructor of this type would be a good place to do so.

    type PublicKey: AsRef<[u8]> + Clone + Eq + Hash;

    /// A private/public key-pair for use in the Diffie-Hellman calculation.

    type KeyPair: KeyPair<PublicKey = Self::PublicKey>;

    /// The result of a Diffie-Hellman calculation.

    type SharedSecret;

    /// A `RootKey` is used in the outer Diffie-Hellman ratchet.

    type RootKey;

    /// A `ChainKey` is used in the inner symmetric ratchets.

    type ChainKey;

    /// A `MessageKey` is used to encrypt/decrypt messages.

    ///

    /// The implementation of this type could be a complex type: for example an implementation that

    /// works by the encrypt-then-MAC paradigm may require a tuple consisting of an encryption key

    /// and a MAC key.

    type MessageKey;

    /// Perform the Diffie-Hellman operation.

    fn diffie\_hellman(us: &Self::KeyPair, them: &Self::PublicKey) -> Self::SharedSecret;

    /// Derive a new root-key/chain-key pair from the old root-key and a fresh shared secret.

    fn kdf\_rk(

        root\_key: &Self::RootKey,

        shared\_secret: &Self::SharedSecret,

    ) -> (Self::RootKey, Self::ChainKey);

    /// Derive a new chain-key/message-key pair from the old chain-key.

    fn kdf\_ck(chain\_key: &Self::ChainKey) -> (Self::ChainKey, Self::MessageKey);

    /// Authenticate-encrypt the plaintext and associated data.

    ///

    /// This method MUST authenticate `associated\_data`, because it contains the header bytes.

    fn encrypt(key: &Self::MessageKey, plaintext: &[u8], associated\_data: &[u8]) -> Vec<u8>;

    /// Verify-decrypt the ciphertext and associated data.

    fn decrypt(

        key: &Self::MessageKey,

        ciphertext: &[u8],

        associated\_data: &[u8],

    ) -> Result<Vec<u8>, DecryptError>;

}

/// A private-/public-key pair

///

/// This trait is required for `CryptoProvider::KeyPair`

pub trait KeyPair {

    /// Type of the public half of the key pair

    ///

    /// This type should be equal to `CryptoProvider::PublicKey`

    type PublicKey;

    /// Generate a new random `KeyPair`

    fn new<R: CryptoRng + RngCore>(rng: &mut R) -> Self;

    /// Get a reference to the public half of the key pair

    fn public(&self) -> &Self::PublicKey;

}

// Maximum amount of skipped message keys that can be stored

const MKS\_CAPACITY: usize = 2000;

// A KeyStore holds the skipped `MessageKey`s.

//

// When messages can arrive out of order, the DoubleRatchet must store the MessageKeys

// corresponding to the messages that were skipped over. See also the [specification] for further

// discussion.

//

// [specification]: https://signal.org/docs/specifications/doubleratchet/#deletion-of-skipped-message-keys

struct KeyStore<CP: CryptoProvider>(HashMap<CP::PublicKey, HashMap<Counter, CP::MessageKey>>);

impl<CP> fmt::Debug for KeyStore<CP>

where

    CP: CryptoProvider,

    CP::PublicKey: fmt::Debug,

    CP::MessageKey: fmt::Debug,

{

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(f, "KeyStore({:?})", self.0)

    }

}

impl<CP: CryptoProvider> KeyStore<CP> {

    fn new() -> Self {

        Self(HashMap::new())

    }

    // Get the MessageKey at `(dh, n)` if it is stored

    fn get(&self, dh: &CP::PublicKey, n: Counter) -> Option<&CP::MessageKey> {

        self.0.get(dh)?.get(&n)

    }

    // Do `n` more MessageKeys fit in the KeyStore?

    fn can\_store(&self, n: usize) -> bool {

        let current: usize = self.0.values().map(HashMap::len).sum();

        current + n <= MKS\_CAPACITY

    }

    // Extend the storage with `mks`

    //

    // Keys are stored at `dh` and `n` counting upwards:

    //   (dh, n  ): mks[0]

    //   (dh, n+1): mks[1]

    //   ...

    fn extend(&mut self, dh: &CP::PublicKey, n: Counter, mks: Vec<CP::MessageKey>) {

        let values = (n..).zip(mks.into\_iter());

        if let Some(v) = self.0.get\_mut(dh) {

            v.extend(values);

        } else {

            self.0.insert(dh.clone(), values.collect());

        }

    }

    // Remove the MessageKey at index `(dh, n)`

    //

    // Assumes the MessageKey is indeed stored.

    fn remove(&mut self, dh: &CP::PublicKey, n: Counter) {

        debug\_assert!(self.0.contains\_key(dh));

        let hm = self.0.get\_mut(dh).unwrap();

        debug\_assert!(hm.contains\_key(&n));

        if hm.len() == 1 {

            self.0.remove(dh);

        } else {

            hm.remove(&n);

        }

    }

}

// Required information for updating the state after successful decryption

enum Diff<CP: CryptoProvider> {

    // Key was found amongst old key

    OldKey,

    // Key was part of the current receive chain

    CurrentChain(CP::ChainKey, Vec<CP::MessageKey>),

    // Key was part of the next receive chain

    NextChain(CP::RootKey, CP::ChainKey, Vec<CP::MessageKey>),

}

/// Error that occurs on `try\_ratchet\_encrypt` before the state is initialized.

#[derive(Clone, Copy, Debug, PartialEq, Eq)]

pub struct EncryptUninit;

#[cfg(feature = "std")]

impl Error for EncryptUninit {}

impl fmt::Display for EncryptUninit {

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(

            f,

            "Encrypt not yet initialized (you must receive a message first)"

        )

    }

}

/// Error that may occur during `ratchet\_decrypt`

#[derive(Clone, Copy, Debug, PartialEq, Eq)]

pub enum DecryptError {

    /// Could not verify-decrypt the ciphertext + associated data + header

    DecryptFailure,

    /// Could not find the message key required for decryption

    ///

    /// Note that this implementation is not always able to detect when an old `MessageKey` can't

    /// be found: a `DecryptFailure` may be triggered instead.

    MessageKeyNotFound,

    /// Header message counter is too large (either `n` or `pn`)

    SkipTooLarge,

    /// Storage of skipped message keys is full

    StorageFull,

}

#[cfg(feature = "std")]

impl Error for DecryptError {}

impl fmt::Display for DecryptError {

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        use DecryptError::\*;

        match self {

            DecryptFailure => write!(f, "Error during verify-decrypting"),

            MessageKeyNotFound => {

                write!(f, "Could not find the message key required for decryption")

            }

            SkipTooLarge => write!(f, "Header message counter is too large"),

            StorageFull => write!(f, "Storage for skipped messages is full"),

        }

    }

}

// Create a mock CryptoProvider for testing purposes. See `tests/signal.rs` for a proper example

// implementation.

// #[cfg(feature = "test")]

#[allow(unused)]

#[allow(missing\_docs)]

pub mod mock {

    use super::\*;

    pub type DoubleRatchet = super::DoubleRatchet<CryptoProvider>;

    pub struct CryptoProvider;

    impl super::CryptoProvider for CryptoProvider {

        type KeyPair = KeyPair;

        type PublicKey = PublicKey;

        type SharedSecret = u8;

        type RootKey = [u8; 2];

        type ChainKey = [u8; 3];

        type MessageKey = [u8; 3];

        fn diffie\_hellman(us: &KeyPair, them: &PublicKey) -> u8 {

            us.0[0].wrapping\_add(them.0[0])

        }

        fn kdf\_rk(rk: &[u8; 2], s: &u8) -> ([u8; 2], [u8; 3]) {

            ([rk[0], \*s], [rk[0], rk[1], 0])

        }

        fn kdf\_ck(ck: &[u8; 3]) -> ([u8; 3], [u8; 3]) {

            ([ck[0], ck[1], ck[2].wrapping\_add(1)], \*ck)

        }

        fn encrypt(mk: &[u8; 3], pt: &[u8], ad: &[u8]) -> Vec<u8> {

            let mut ct = Vec::from(&mk[..]);

            ct.extend\_from\_slice(pt);

            ct.extend\_from\_slice(ad);

            ct

        }

        fn decrypt(mk: &[u8; 3], ct: &[u8], ad: &[u8]) -> Result<Vec<u8>, super::DecryptError> {

            if ct.len() < 3 + ad.len() || ct[..3] != mk[..] || !ct.ends\_with(ad) {

                Err(super::DecryptError::DecryptFailure)

            } else {

                Ok(Vec::from(&ct[3..ct.len() - ad.len()]))

            }

        }

    }

    #[derive(Clone, Debug, Hash, PartialEq, Eq)]

    pub struct PublicKey([u8; 1]);

    impl AsRef<[u8]> for PublicKey {

        fn as\_ref(&self) -> &[u8] {

            &self.0

        }

    }

    #[derive(Debug)]

    pub struct KeyPair([u8; 1], PublicKey);

    impl super::KeyPair for KeyPair {

        type PublicKey = PublicKey;

        #[allow(clippy::cast\_possible\_truncation)]

        fn new<R: rand\_core::CryptoRng + rand\_core::RngCore>(rng: &mut R) -> Self {

            let n = rng.next\_u32() as u8;

            Self([n], PublicKey([n + 1]))

        }

        fn public(&self) -> &PublicKey {

            &self.1

        }

    }

    // FIXME: [sebastianv89] this functionality exists already, but

    // breaks the build... use rand::rngs::mock::StepRng;

    #[derive(Default)]

    pub struct Rng(u64);

    impl rand\_core::RngCore for Rng {

        fn next\_u64(&mut self) -> u64 {

            self.0 += 1;

            self.0

        }

        #[allow(clippy::cast\_possible\_truncation)]

        fn next\_u32(&mut self) -> u32 {

            self.next\_u64() as u32

        }

        fn fill\_bytes(&mut self, out: &mut [u8]) {

            rand\_core::impls::fill\_bytes\_via\_next(self, out);

        }

        fn try\_fill\_bytes(&mut self, out: &mut [u8]) -> Result<(), rand\_core::Error> {

            self.fill\_bytes(out);

            Ok(())

        }

    }

    impl super::CryptoRng for Rng {}

}

#[cfg(test)]

mod tests {

    use super::\*;

    type DR = DoubleRatchet<mock::CryptoProvider>;

    fn asymmetric\_setup(rng: &mut mock::Rng) -> (DR, DR) {

        let secret = [42, 0];

        let pair = mock::KeyPair::new(rng);

        let pubkey = pair.public().clone();

        let alice = DR::new\_alice(&secret, pubkey, None, rng);

        let bob = DR::new\_bob(secret, pair, None);

        (alice, bob)

    }

    fn symmetric\_setup(rng: &mut mock::Rng) -> (DR, DR) {

        let secret = [42, 0];

        let ck\_init = [42, 0, 0];

        let pair = mock::KeyPair::new(rng);

        let pubkey = pair.public().clone();

        let alice = DR::new\_alice(&secret, pubkey, Some(ck\_init), rng);

        let bob = DR::new\_bob(secret, pair, Some(ck\_init));

        (alice, bob)

    }

    #[test]

    fn test\_asymmetric\_setup() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        // Alice can encrypt, Bob can't

        let (pt\_a, ad\_a) = (b"Hi Bobby", b"A2B");

        let (pt\_b, ad\_b) = (b"What's up Al?", b"B2A");

        let (h\_a, ct\_a) = alice.ratchet\_encrypt(pt\_a, ad\_a, &mut rng);

        assert\_eq!(

            Err(EncryptUninit),

            bob.try\_ratchet\_encrypt(pt\_b, ad\_b, &mut rng)

        );

        assert\_eq!(

            Ok(Vec::from(&pt\_a[..])),

            bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a)

        );

        // but after decryption Bob can encrypt

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(pt\_b, ad\_b, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_b[..])),

            alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b)

        );

    }

    #[test]

    fn test\_symmetric\_setup() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = symmetric\_setup(&mut rng);

        // Alice can encrypt, Bob can't

        let (pt\_a, ad\_a) = (b"Hi Bobby", b"A2B");

        let (pt\_b, ad\_b) = (b"What's up Al?", b"B2A");

        let (h\_a, ct\_a) = alice.ratchet\_encrypt(pt\_a, ad\_a, &mut rng);

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(pt\_b, ad\_b, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_a[..])),

            bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a)

        );

        assert\_eq!(

            Ok(Vec::from(&pt\_b[..])),

            alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b)

        );

    }

    #[test]

    fn symmetric\_out\_of\_order() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        // Alice's message arrive out of order, some are even missing

        let pt\_a\_0 = b"Hi Bobby";

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(pt\_a\_0, ad\_a, &mut rng);

        for \_ in 1..9 {

            alice.ratchet\_encrypt(b"hello?", ad\_a, &mut rng); // drop these messages

        }

        let pt\_a\_9 = b"are you there?";

        let (h\_a\_9, ct\_a\_9) = alice.ratchet\_encrypt(pt\_a\_9, ad\_a, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_a\_9[..])),

            bob.ratchet\_decrypt(&h\_a\_9, &ct\_a\_9, ad\_a)

        );

        assert\_eq!(

            Ok(Vec::from(&pt\_a\_0[..])),

            bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a)

        );

        // Bob's replies also arrive out of order

        let pt\_b\_0 = b"Yes I'm here";

        let (h\_b\_0, ct\_b\_0) = bob.ratchet\_encrypt(pt\_b\_0, ad\_b, &mut rng);

        for \_ in 1..9 {

            bob.ratchet\_encrypt(b"why?", ad\_b, &mut rng); // drop these messages

        }

        let pt\_b\_9 = b"Tell me why!!!";

        let (h\_b\_9, ct\_b\_9) = bob.ratchet\_encrypt(pt\_b\_9, ad\_b, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_b\_9[..])),

            alice.ratchet\_decrypt(&h\_b\_9, &ct\_b\_9, ad\_b)

        );

        assert\_eq!(

            Ok(Vec::from(&pt\_b\_0[..])),

            alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b)

        );

    }

    #[test]

    fn dh\_out\_of\_order() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        let pt\_a\_0 = b"Good day Robert";

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(pt\_a\_0, ad\_a, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_a\_0[..])),

            bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a)

        );

        let pt\_a\_1 = b"Do you like Rust?";

        let (h\_a\_1, ct\_a\_1) = alice.ratchet\_encrypt(pt\_a\_1, ad\_a, &mut rng);

        // Bob misses pt\_a\_1

        let pt\_b\_0 = b"Salutations Allison";

        let (h\_b\_0, ct\_b\_0) = bob.ratchet\_encrypt(pt\_b\_0, ad\_b, &mut rng);

        // Alice misses pt\_b\_0

        let pt\_b\_1 = b"How is your day going?";

        let (h\_b\_1, ct\_b\_1) = bob.ratchet\_encrypt(pt\_b\_1, ad\_b, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_b\_1[..])),

            alice.ratchet\_decrypt(&h\_b\_1, &ct\_b\_1, ad\_b)

        );

        let pt\_a\_2 = b"My day is fine.";

        let (h\_a\_2, ct\_a\_2) = alice.ratchet\_encrypt(pt\_a\_2, ad\_a, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_a\_2[..])),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2, ad\_a)

        );

        // now Bob receives pt\_a\_1

        assert\_eq!(

            Ok(Vec::from(&pt\_a\_1[..])),

            bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a)

        );

        let pt\_b\_2 = b"Yes I like Rust";

        let (h\_b\_2, ct\_b\_2) = bob.ratchet\_encrypt(pt\_b\_2, ad\_b, &mut rng);

        assert\_eq!(

            Ok(Vec::from(&pt\_b\_2[..])),

            alice.ratchet\_decrypt(&h\_b\_2, &ct\_b\_2, ad\_b)

        );

        // now Alice receives pt\_b\_0

        assert\_eq!(

            Ok(Vec::from(&pt\_b\_0[..])),

            alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b)

        );

    }

    #[test]

    #[should\_panic(expected = "not yet initialized for encryption")]

    fn encrypt\_error() {

        let mut rng = mock::Rng::default();

        let (\_alice, mut bob) = asymmetric\_setup(&mut rng);

        assert\_eq!(

            Err(EncryptUninit),

            bob.try\_ratchet\_encrypt(b"", b"", &mut rng)

        );

        bob.ratchet\_encrypt(b"", b"", &mut rng);

    }

    #[test]

    fn decrypt\_failure() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        // Next chain

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        let mut ct\_a\_0\_err = ct\_a\_0.clone();

        ct\_a\_0\_err[2] ^= 0x80;

        let mut h\_a\_0\_err = h\_a\_0.clone();

        h\_a\_0\_err.pn = 1;

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0\_err, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_0\_err, &ct\_a\_0, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_b)

        );

        // Current Chain

        let (h\_a\_1, ct\_a\_1) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a).unwrap();

        let (h\_a\_2, ct\_a\_2) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        let mut h\_a\_2\_err = h\_a\_2.clone();

        h\_a\_2\_err.pn += 1;

        let mut ct\_a\_2\_err = ct\_a\_2.clone();

        ct\_a\_2\_err[0] ^= 0x04;

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2\_err, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2\_err, &ct\_a\_2, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2, ad\_b)

        );

        // Previous chain

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(b"Hi Alice", ad\_b, &mut rng);

        alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b).unwrap();

        let (h\_a\_3, ct\_a\_3) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        bob.ratchet\_decrypt(&h\_a\_3, &ct\_a\_3, ad\_a).unwrap();

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2\_err, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2\_err, &ct\_a\_2, ad\_a)

        );

        assert\_eq!(

            Err(DecryptError::DecryptFailure),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2, ad\_b)

        );

    }

    #[test]

    fn double\_sending() {

        // The implementation is unable to consistently detect why decryption fails when receiving

        // double messages: the only requirement should be that \*any\* error is triggered.

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(b"Whatever", ad\_a, &mut rng);

        bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a).unwrap();

        assert!(bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a).is\_err());

        let (h\_b\_0, ct\_b\_0) = bob.ratchet\_encrypt(b"Whatever", ad\_b, &mut rng);

        alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b).unwrap();

        assert!(alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b).is\_err());

        let (h\_a\_1, ct\_a\_1) = alice.ratchet\_encrypt(b"Whatever", ad\_a, &mut rng);

        bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a).unwrap();

        assert!(bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a).is\_err());

        let (h\_b\_1, ct\_b\_1) = bob.ratchet\_encrypt(b"Whatever", ad\_b, &mut rng);

        alice.ratchet\_decrypt(&h\_b\_1, &ct\_b\_1, ad\_b).unwrap();

        assert!(alice.ratchet\_decrypt(&h\_b\_1, &ct\_b\_1, ad\_b).is\_err());

        assert!(bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a).is\_err());

        assert!(alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b).is\_err());

    }

    #[test]

    fn invalid\_header() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a).unwrap();

        let (h\_b\_0, ct\_b\_0) = bob.ratchet\_encrypt(b"Hi Alice", ad\_b, &mut rng);

        alice.ratchet\_decrypt(&h\_b\_0, &ct\_b\_0, ad\_b).unwrap();

        let (mut h\_a\_1, ct\_a\_1) = alice.ratchet\_encrypt(b"I will lie to you now", ad\_a, &mut rng);

        assert\_eq!(h\_a\_1.pn, 1);

        h\_a\_1.pn = 0;

        assert!(bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a).is\_err());

    }

    #[test]

    fn skip\_too\_large() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        let (h\_a\_0, ct\_a\_0) = alice.ratchet\_encrypt(b"Hi Bob", ad\_a, &mut rng);

        for \_ in 0..=MAX\_SKIP {

            alice.ratchet\_encrypt(b"Not sending this", ad\_a, &mut rng);

        }

        let (h\_a\_1, ct\_a\_1) = alice.ratchet\_encrypt(b"n > MAXSKIP", ad\_a, &mut rng);

        assert\_eq!(

            Err(DecryptError::SkipTooLarge),

            bob.ratchet\_decrypt(&h\_a\_1, &ct\_a\_1, ad\_a)

        );

        bob.ratchet\_decrypt(&h\_a\_0, &ct\_a\_0, ad\_a).unwrap();

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(b"Hi Alice", ad\_b, &mut rng);

        alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b).unwrap();

        let (h\_a\_2, ct\_a\_2) = alice.ratchet\_encrypt(b"pn > MAXSKIP", ad\_a, &mut rng);

        assert\_eq!(

            Err(DecryptError::SkipTooLarge),

            bob.ratchet\_decrypt(&h\_a\_2, &ct\_a\_2, ad\_a)

        );

    }

    #[test]

    fn storage\_full() {

        let mut rng = mock::Rng::default();

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        let ad\_a = b"A2B";

        let mut stored = 0;

        while stored < MKS\_CAPACITY {

            for \_ in 0..cmp::min(MAX\_SKIP, MKS\_CAPACITY - stored) {

                alice.ratchet\_encrypt(b"Not sending this", ad\_a, &mut rng);

            }

            let (h\_a, ct\_a) = alice.ratchet\_encrypt(b"Hello Bob", ad\_a, &mut rng);

            bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a).unwrap();

            stored += MAX\_SKIP;

            let \_ = &bob.mkskipped.0.values().map(|hm| hm.len()).sum::<usize>();

        }

        alice.ratchet\_encrypt(b"Bob can't store this key anymore", ad\_a, &mut rng);

        let (h\_a, ct\_a) = alice.ratchet\_encrypt(b"Gotcha, Bob!", ad\_a, &mut rng);

        assert\_eq!(

            Err(DecryptError::StorageFull),

            bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a)

        );

    }

    #[test]

    fn cannot\_crash\_other() {

        // Malicious parties should not be able to crash the other end (this was an

        // issue in an old implementation).

        let mut rng = mock::Rng::default();

        let (ad\_a, ad\_b) = (b"A2B", b"B2A");

        let (mut alice, mut bob) = symmetric\_setup(&mut rng);

        alice.pn = 10;

        bob.pn = 10;

        let (h\_a, ct\_a) = alice.ratchet\_encrypt(b"not important", ad\_a, &mut rng);

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(b"not important", ad\_b, &mut rng);

        let \_ = alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b);

        let \_ = bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a);

        let (mut alice, mut bob) = asymmetric\_setup(&mut rng);

        alice.pn = 10;

        let (h\_a, ct\_a) = alice.ratchet\_encrypt(b"not important", ad\_a, &mut rng);

        let \_ = bob.ratchet\_decrypt(&h\_a, &ct\_a, ad\_a);

        bob.pn = 10;

        let (h\_b, ct\_b) = bob.ratchet\_encrypt(b"not important", ad\_b, &mut rng);

        let \_ = alice.ratchet\_decrypt(&h\_b, &ct\_b, ad\_b);

    }

}

Приложение 2. Реализация сервера для обмена сообщениями.

use std::io::{ ErrorKind, Read, Write };

use std::net::TcpListener;

use std::sync::mpsc;

use std::thread;

use std::sync::mpsc::{ Receiver, Sender };

use std::time::SystemTime;

use std::net::TcpStream;

use chrono::{ DateTime, Utc };

// Add these struct definitions at the top

struct Client {

    socket: TcpStream,

    connected\_at: SystemTime,

}

struct Message {

    data: Vec<u8>,

    timestamp: SystemTime,

}

const LOCAL: &str = "127.0.0.1:6000";

const MSG\_SIZE: usize = 512;

fn sleep() {

    thread::sleep(::std::time::Duration::from\_millis(100));

}

fn main() {

    let server = TcpListener::bind(LOCAL).expect("Listener failed to bind");

    server.set\_nonblocking(true).expect("failed to initialize non-blocking");

    let mut clients: Vec<Client> = vec![];

    let (tx, rx): (Sender<Vec<u8>>, Receiver<Vec<u8>>) = mpsc::channel();

    loop {

        if let Ok((socket, addr)) = server.accept() {

            println!("Client {} connected", addr);

            let tx = tx.clone();

            let mut client\_socket = socket.try\_clone().expect("Failed to clone client socket");

            clients.push(Client {

                socket,

                connected\_at: SystemTime::now(),

            });

            thread::spawn(move || {

                loop {

                    let mut buff = vec![0; MSG\_SIZE];

                    match client\_socket.read\_exact(&mut buff) {

                        Ok(\_) => {

                            let total\_len = u32::from\_le\_bytes(

                                buff[0..4].try\_into().unwrap()

                            ) as usize;

                            if total\_len > MSG\_SIZE {

                                let mut rest = vec![0; total\_len - MSG\_SIZE];

                                match client\_socket.read\_exact(&mut rest) {

                                    Ok(\_) => {

                                        buff.extend(rest);

                                    }

                                    Err(e) => {

                                        println!("failed to read rest of message: {:?}", e);

                                        break;

                                    }

                                }

                            }

                            let msg = Message {

                                data: buff.clone(),

                                timestamp: SystemTime::now(),

                            };

                            let subsec\_nanos = msg.timestamp

                                .duration\_since(std::time::UNIX\_EPOCH)

                                .unwrap()

                                .subsec\_nanos();

                            let ms = subsec\_nanos / 1\_000\_000;

                            let ns = subsec\_nanos % 1\_000\_000;

                            let timestamp = msg.timestamp

                                .duration\_since(std::time::UNIX\_EPOCH)

                                .unwrap()

                                .as\_secs() as i64;

                            let datetime = DateTime::<Utc>::from\_timestamp(timestamp, 0).unwrap();

                            let full\_msg\_hex = buff.iter().map(|b| format!("{:02X}", b)).collect::<String>()[..total\_len \* 2].to\_string();

                            println!(

                                "{}: {}.{:03}.{:06} UTC transmitted {} bytes: {}",

                                addr,

                                datetime.format("%Y-%m-%d %H:%M:%S"),

                                ms,

                                ns,

                                total\_len,

                                full\_msg\_hex

                            );

                            tx.send(msg.data).expect("failed to send msg to rx");

                        }

                        Err(ref err) if err.kind() == ErrorKind::WouldBlock => (),

                        Err(\_) => {

                            println!("closing connection with: {}", addr);

                            break;

                        }

                    }

                    sleep();

                }

            });

        }

        if let Ok(msg) = rx.try\_recv() {

            clients = clients

                .into\_iter()

                .filter\_map(|mut client| {

                    let mut buff = msg.clone();

                    let total\_len = u32::from\_le\_bytes(buff[0..4].try\_into().unwrap()) as usize;

                    let packet\_len = std::cmp::max(total\_len, MSG\_SIZE);

                    buff.resize(packet\_len, 0);

                    if client.connected\_at <= SystemTime::now() {

                        client.socket

                            .write\_all(&buff)

                            .map(|\_| client)

                            .ok()

                    } else {

                        Some(client)

                    }

                })

                .collect::<Vec<\_>>();

        }

        sleep();

    }

}

Приложение 3. Реализация клиента для обмена сообщениями.

use aes::cipher::{ BlockSizeUser, KeySizeUser };

use aes::Aes256;

use block\_padding::Pkcs7;

use cbc;

use cipher::{ BlockDecryptMut, BlockEncryptMut, KeyInit, KeyIvInit };

use clear\_on\_drop::clear::Clear;

use generic\_array::{ typenum::U32, GenericArray };

use hkdf::Hkdf;

use hmac::digest::OutputSizeUser;

use hmac::{ Hmac, Mac };

use ksi\_double\_ratchet::{ self as dr, Header, KeyPair as \_ };

use rand\_core::{ CryptoRng, RngCore };

use rand\_os::OsRng;

use sha2::Sha256;

use std::fmt;

use std::hash::{ Hash, Hasher };

use std::io::{ self, ErrorKind, Read, Write };

use std::net::TcpStream;

use std::sync::mpsc::{ self, TryRecvError };

use std::thread;

use std::time::Duration;

use x25519\_dalek::{ self, SharedSecret };

use base64::prelude::\*;

const SERVER: &str = "127.0.0.1:6000";

const BASE\_MSG\_SIZE: usize = 512;

pub type SignalDR = dr::DoubleRatchet<SignalCryptoProvider>;

pub struct SignalCryptoProvider;

impl dr::CryptoProvider for SignalCryptoProvider {

    type PublicKey = PublicKey;

    type KeyPair = KeyPair;

    type SharedSecret = SharedSecret;

    type RootKey = SymmetricKey;

    type ChainKey = SymmetricKey;

    type MessageKey = SymmetricKey;

    fn diffie\_hellman(us: &KeyPair, them: &PublicKey) -> SharedSecret {

        us.private.diffie\_hellman(&them.0)

    }

    fn kdf\_rk(rk: &SymmetricKey, s: &SharedSecret) -> (SymmetricKey, SymmetricKey) {

        let salt: Option<&[u8]> = Some(rk.0.as\_slice());

        let input\_key\_material: &[u8; 32] = s.as\_bytes();

        let pseudo\_random\_key = Hkdf::<Sha256>::new(salt, input\_key\_material);

        let info = &b"WhisperRatchet"[..];

        let mut output\_key\_material = [0; 64];

        pseudo\_random\_key.expand(&info, &mut output\_key\_material).unwrap();

        let root\_key: GenericArray<u8, \_> = GenericArray::<u8, U32>

            ::from\_slice(&output\_key\_material[..32])

            .clone();

        let chain\_key: GenericArray<u8, \_> = GenericArray::<u8, U32>

            ::from\_slice(&output\_key\_material[32..])

            .clone();

        return (SymmetricKey(root\_key), SymmetricKey(chain\_key));

    }

    fn kdf\_ck(ck: &SymmetricKey) -> (SymmetricKey, SymmetricKey) {

        let key = ck.0.as\_slice();

        let mut mac = <Hmac<Sha256> as KeyInit>::new\_from\_slice(key).unwrap();

        mac.update(&[0x01]);

        let message\_key = mac.finalize().into\_bytes();

        // TODO: check if this is correct, maybe we should not recreate the mac and just somehow clear it

        //      - Kseen715, 2024.11.29

        mac = <Hmac<Sha256> as KeyInit>::new\_from\_slice(key).unwrap();

        mac.update(&[0x02]);

        let chain\_key = mac.finalize().into\_bytes();

        return (SymmetricKey(chain\_key), SymmetricKey(message\_key));

    }

    fn encrypt(key: &SymmetricKey, pt: &[u8], ad: &[u8]) -> Vec<u8> {

        let input\_key\_material = key.0.as\_slice();

        let pseudo\_random\_key = Hkdf::<Sha256>::new(None, input\_key\_material);

        let info = b"WhisperMessageKeys";

        let mut output\_key\_material = [0; 80];

        pseudo\_random\_key.expand(info, &mut output\_key\_material).unwrap();

        let encryption\_key = GenericArray::<u8, <Aes256 as KeySizeUser>::KeySize>::from\_slice(

            &output\_key\_material[..32]

        );

        let message\_key = GenericArray::<

            u8,

            <Hmac<Sha256> as OutputSizeUser>::OutputSize

        >::from\_slice(&output\_key\_material[32..64]);

        let iv = GenericArray::<u8, <Aes256 as BlockSizeUser>::BlockSize>::from\_slice(

            &output\_key\_material[64..]

        );

        type Aes256CbcEnc = cbc::Encryptor<Aes256>;

        let ciphr = Aes256CbcEnc::new(encryption\_key, iv);

        let mut mod\_plaintext = pt.to\_vec();

        let plaintext\_len = pt.len();

        mod\_plaintext.resize(plaintext\_len + 16 - (plaintext\_len % 16), 0);

        let ciphertext = match ciphr.encrypt\_padded\_mut::<Pkcs7>(&mut mod\_plaintext, plaintext\_len) {

            Ok(encrypted) => encrypted,

            Err(e) => panic!("Error: {:?}", e),

        };

        let mut mac = <Hmac<Sha256> as Mac>::new\_from\_slice(message\_key).unwrap();

        mac.update(ad);

        mac.update(&ciphertext);

        let tag = mac.finalize().into\_bytes();

        let mut ciphertext\_vec = ciphertext.to\_vec();

        ciphertext\_vec.extend((&tag[..8]).into\_iter());

        output\_key\_material.clear();

        // println!("ct: {:?}", ciphertext\_vec);

        ciphertext\_vec

    }

    fn decrypt(key: &SymmetricKey, ct: &[u8], ad: &[u8]) -> Result<Vec<u8>, dr::DecryptError> {

        let input\_key\_material = key.0.as\_slice();

        let pseudo\_random\_key = Hkdf::<Sha256>::new(None, input\_key\_material);

        let info = b"WhisperMessageKeys";

        let mut output\_key\_material = [0; 80];

        match pseudo\_random\_key.expand(info, &mut output\_key\_material) {

            Ok(\_) => (),

            Err(e) => panic!("Error: {:?}", e),

        }

        let decryption\_key = GenericArray::<u8, <Aes256 as KeySizeUser>::KeySize>::from\_slice(

            &output\_key\_material[..32]

        );

        let message\_key = GenericArray::<

            u8,

            <Hmac<Sha256> as OutputSizeUser>::OutputSize

        >::from\_slice(&output\_key\_material[32..64]);

        let iv = GenericArray::<u8, <Aes256 as BlockSizeUser>::BlockSize>::from\_slice(

            &output\_key\_material[64..]

        );

        let ciphertext\_len = ct.len() - 8;

        let mut mac = match <Hmac<Sha256> as Mac>::new\_from\_slice(message\_key) {

            Ok(mac) => mac,

            Err(e) => panic!("Error: {:?}", e),

        };

        mac.update(ad);

        mac.update(&ct[..ciphertext\_len]);

        let tag = mac.finalize().into\_bytes();

        if bool::from(!(&tag[..8] == &ct[ciphertext\_len..])) {

            output\_key\_material.clear();

            println!("Error: {:?}", dr::DecryptError::DecryptFailure);

            return Err(dr::DecryptError::DecryptFailure);

        }

        type Aes256CbcDec = cbc::Decryptor<Aes256>;

        let ciphr = Aes256CbcDec::new(decryption\_key, iv);

        let mut ciphertext\_vec = ct[..ciphertext\_len].to\_vec();

        return match ciphr.decrypt\_padded\_mut::<Pkcs7>(&mut ciphertext\_vec) {

            Ok(pt) => {

                output\_key\_material.clear();

                // println!("pt: {:?}", pt);

                Ok(pt.to\_vec())

            }

            Err(e) => {

                output\_key\_material.clear();

                println!("Error: {:?}", e);

                Err(dr::DecryptError::DecryptFailure)

            }

        };

    }

}

#[derive(Clone, Debug)]

pub struct PublicKey(x25519\_dalek::PublicKey);

impl Eq for PublicKey {}

impl PartialEq for PublicKey {

    fn eq(&self, other: &PublicKey) -> bool {

        self.0.as\_bytes() == other.0.as\_bytes()

    }

}

impl Hash for PublicKey {

    fn hash<H: Hasher>(&self, state: &mut H) {

        self.0.as\_bytes().hash(state);

    }

}

impl<'a> From<&'a x25519\_dalek::StaticSecret> for PublicKey {

    fn from(private: &'a x25519\_dalek::StaticSecret) -> PublicKey {

        PublicKey(x25519\_dalek::PublicKey::from(private))

    }

}

impl AsRef<[u8]> for PublicKey {

    fn as\_ref(&self) -> &[u8] {

        self.0.as\_bytes()

    }

}

pub struct KeyPair {

    private: x25519\_dalek::StaticSecret,

    public: PublicKey,

}

impl fmt::Debug for KeyPair {

    #[cfg(debug\_assertions)]

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(

            f,

            "KeyPair {{ private (bytes): {:?}, public: {:?} }}",

            self.private.to\_bytes(),

            self.public

        )

    }

    #[cfg(not(debug\_assertions))]

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(f, "KeyPair {{ private (bytes): <hidden bytes>, public: {:?} }}", self.public)

    }

}

impl dr::KeyPair for KeyPair {

    type PublicKey = PublicKey;

    fn new<R: CryptoRng + RngCore>(rng: &mut R) -> KeyPair {

        let private = x25519\_dalek::StaticSecret::new(rng);

        let public = PublicKey::from(&private);

        KeyPair { private, public }

    }

    fn public(&self) -> &PublicKey {

        &self.public

    }

}

#[derive(Default)]

pub struct SymmetricKey(GenericArray<u8, U32>);

impl fmt::Debug for SymmetricKey {

    #[cfg(debug\_assertions)]

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(f, "SymmetricKey({:?})", self.0)

    }

    #[cfg(not(debug\_assertions))]

    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

        write!(f, "SymmetricKey(<hidden bytes>)")

    }

}

impl Drop for SymmetricKey {

    fn drop(&mut self) {

        self.0.clear();

    }

}

impl Clone for SymmetricKey {

    fn clone(&self) -> SymmetricKey {

        SymmetricKey(self.0.clone())

    }

}

impl Clone for KeyPair {

    fn clone(&self) -> KeyPair {

        KeyPair {

            private: self.private.clone(),

            public: self.public.clone(),

        }

    }

}

impl AsRef<[u8]> for SymmetricKey {

    fn as\_ref(&self) -> &[u8] {

        self.0.as\_slice()

    }

}

fn read\_msg\_from\_buff(

    buff: &Vec<u8>,

    our\_auth\_data\_b64: &String,

    bob\_auth\_data\_b64: &String

) -> (String, String, String) {

    // println!("[dbg] Start read\_msg\_from\_buff");

    let total\_len = u32::from\_le\_bytes(buff[0..4].try\_into().unwrap()) as usize;

    // println!("total\_len: {:?}", total\_len);

    // let my\_session\_auth\_data = our\_auth\_data\_b64.clone();

    let meta\_len = u32::from\_le\_bytes(buff[4..8].try\_into().unwrap()) as usize;

    let meta = &buff[8..8 + meta\_len];

    // println!("meta\_len: {:?}", meta\_len);

    let sent\_from\_auth\_data\_len = u32::from\_le\_bytes(

        meta[0..4].to\_vec().try\_into().unwrap()

    ) as usize;

    // println!("[312] sent\_from\_auth\_data\_len: {:?}", sent\_from\_auth\_data\_len);

    let sent\_from\_auth\_data = meta[4..sent\_from\_auth\_data\_len + 4].to\_vec();

    let sent\_from\_auth\_data = String::from\_utf8(sent\_from\_auth\_data).expect(

        "Invalid utf8 sent\_from\_auth\_data"

    );

    let sent\_to\_auth\_data\_len = u32::from\_le\_bytes(

        meta[sent\_from\_auth\_data\_len + 4..sent\_from\_auth\_data\_len + 8].to\_vec().try\_into().unwrap()

    ) as usize;

    // println!("sent\_to\_auth\_data\_len: {:?}", sent\_to\_auth\_data\_len);

    let sent\_to\_auth\_data =

        meta[

            sent\_from\_auth\_data\_len + 8..sent\_from\_auth\_data\_len + 8 + sent\_to\_auth\_data\_len

        ].to\_vec();

    let sent\_to\_auth\_data = String::from\_utf8(sent\_to\_auth\_data).expect(

        "Invalid utf8 sent\_to\_auth\_data"

    );

    let header\_len: usize = u32::from\_le\_bytes(

        meta[

            sent\_from\_auth\_data\_len + 8 + sent\_to\_auth\_data\_len..sent\_from\_auth\_data\_len +

                12 +

                sent\_to\_auth\_data\_len

        ]

            .to\_vec()

            .try\_into()

            .unwrap()

    ) as usize;

    // println!("header\_len: {:?}", header\_len);

    let header =

        meta[

            sent\_from\_auth\_data\_len + 12 + sent\_to\_auth\_data\_len..sent\_from\_auth\_data\_len +

                12 +

                sent\_to\_auth\_data\_len +

                header\_len

        ].to\_vec();

    // println!("header\_b64: {:?}", header);

    let header = String::from\_utf8(header).expect("Invalid utf8 header");

    // println!("header: {:?}", header);

    // println!("sent\_from\_auth\_data\_b64: {:?}", sent\_from\_auth\_data);

    // println!("sent\_to\_auth\_data\_b64: {:?}", sent\_to\_auth\_data);

    // println!("our\_auth\_data\_b64: {:?}", our\_auth\_data\_b64);

    // println!("bob\_auth\_data\_b64: {:?}", bob\_auth\_data\_b64);

    // println!("calc\_size: {:?}", total\_len - (8 + meta\_len));

    let msg\_text = buff[8 + meta\_len..total\_len].to\_vec();

    let msg\_text = String::from\_utf8(msg\_text).expect("Invalid utf8 message");

    // println!("msg\_text\_b64: {:?}", msg\_text);

    if

        sent\_to\_auth\_data != our\_auth\_data\_b64.clone() ||

        sent\_from\_auth\_data != bob\_auth\_data\_b64.clone()

    {

        // drop the message if it's not from the person we're talking to

        // or if it's from ourselves

        // println!("[dbg] End read\_msg\_from\_buff");

        return (String::from(""), String::from(""), String::from(""));

    }

    // println!("[dbg] End read\_msg\_from\_buff");

    return (sent\_from\_auth\_data, header, msg\_text);

}

fn prepare\_buff\_to\_send\_msg(

    buff: &mut Vec<u8>,

    our\_auth\_data\_b64: &String,

    bob\_auth\_data\_b64: &String,

    header\_b64: &String,

    msg\_b64: &String

) {

    // println!("[dbg] Start prepare\_buff\_to\_send\_msg");

    let msg\_b64\_text\_len = msg\_b64.len();

    // println!("our\_auth\_data\_b64: {:?}", our\_auth\_data\_b64);

    // println!("bob\_auth\_data\_b64: {:?}", bob\_auth\_data\_b64);

    // println!("header\_b64: {:?}", header\_b64);

    // println!("msg\_b64: {:?}", msg\_b64);

    let session\_auth\_data = our\_auth\_data\_b64.clone();

    let session\_auth\_data\_len: u32 = session\_auth\_data.len() as u32;

    let bobs\_auth\_data = bob\_auth\_data\_b64.clone();

    let bobs\_auth\_data\_len: u32 = bobs\_auth\_data.len() as u32;

    let header\_b64\_len = header\_b64.len() as u32;

    let meta\_len: u32 =

        4 + // session\_auth\_data\_len

        session\_auth\_data\_len +

        4 + // bobs\_auth\_data\_len

        bobs\_auth\_data\_len +

        4 + // header\_len

        (header\_b64\_len as u32);

    let total\_len: u32 =

        4 + // total\_len

        4 + // meta\_len

        meta\_len +

        (msg\_b64\_text\_len as u32);

    // println!("total\_len: {:?}", total\_len);

    // println!("meta\_len: {:?}", meta\_len);

    // println!("session\_auth\_data\_len: {:?}", session\_auth\_data\_len);

    // println!("bobs\_auth\_data\_len: {:?}", bobs\_auth\_data\_len);

    // println!("header\_b64\_len: {:?}", header\_b64\_len);

    // println!("msg\_b64\_text\_len: {:?}", msg\_b64\_text\_len);

    buff.extend\_from\_slice(&total\_len.to\_le\_bytes());

    buff.extend\_from\_slice(&meta\_len.to\_le\_bytes());

    // meta:

    buff.extend\_from\_slice(&session\_auth\_data\_len.to\_le\_bytes());

    buff.append(&mut session\_auth\_data.clone().into\_bytes());

    buff.extend\_from\_slice(&bobs\_auth\_data\_len.to\_le\_bytes());

    buff.append(&mut bobs\_auth\_data.clone().into\_bytes());

    buff.extend\_from\_slice(&header\_b64\_len.to\_le\_bytes());

    buff.append(&mut header\_b64.clone().into\_bytes());

    // msg\_b64\_text:

    buff.append(&mut msg\_b64.clone().into\_bytes());

    let packet\_len = std::cmp::max(BASE\_MSG\_SIZE, buff.len());

    buff.resize(packet\_len, 0);

    // println!("[dbg] End prepare\_buff\_to\_send\_msg");

}

// MSG arch:

//      - 4u8: total\_len

//      - 4u8: meta\_len

//          - 4u8: session\_auth\_data\_len

//              - session\_auth\_data

//          - 4u8: bobs\_auth\_data\_len

//              - bobs\_auth\_data

//          - 4u8: header\_len

//              - header

//      - msg

fn main() {

    let mut rng = OsRng::new().unwrap();

    let shared\_key: SymmetricKey = SymmetricKey(

        GenericArray::<u8, U32>::clone\_from\_slice(b"Output of a X3DH key exchange...")

    );

    let mut client = TcpStream::connect(SERVER).expect("Stream failed to connect");

    client.set\_nonblocking(true).expect("failed to initiate non-blocking");

    let (tx, rx) = mpsc::channel::<String>();

    fn read\_input(prompt: &str) -> String {

        print!("{}", prompt);

        io::stdout().flush().expect("Failed to flush stdout");

        let mut input = String::new();

        io::stdin().read\_line(&mut input).expect("Failed to read input");

        input.trim().to\_string()

    }

    let mut session\_created: bool = false;

    #[derive(Debug)]

    enum UserKind {

        Undefined,

        Alice,

        Bob,

    }

    impl PartialEq for UserKind {

        fn eq(&self, other: &UserKind) -> bool {

            match (self, other) {

                (UserKind::Undefined, UserKind::Undefined) => true,

                (UserKind::Alice, UserKind::Alice) => true,

                (UserKind::Bob, UserKind::Bob) => true,

                \_ => false,

            }

        }

    }

    fn bytes\_to\_type<T>(bytes: &[u8]) -> Header<PublicKey> {

        let mut value = std::mem::MaybeUninit::uninit();

        // println!("bytes: {:?}", bytes);

        // println!("size of T: {:?}", std::mem::size\_of::<T>());

        // println!("size of bytes: {:?}", bytes.len());

        unsafe {

            // bytes is raw bytes of the struct

            std::ptr::copy\_nonoverlapping(

                bytes.as\_ptr(),

                value.as\_mut\_ptr() as \*mut u8,

                std::mem::size\_of::<T>()

            );

            // Some()

            value.assume\_init()

        }

    }

    let mut user\_kind: UserKind = UserKind::Undefined;

    let mut alice: SignalDR = SignalDR::new\_alice(

        &shared\_key,

        PublicKey(x25519\_dalek::PublicKey::from([0; 32])),

        None,

        &mut rng

    );

    let mut bob: SignalDR = SignalDR::new\_bob(shared\_key, KeyPair::new(&mut rng), None);

    let our\_auth\_data: String = read\_input("Input your auth data: ");

    let bob\_auth\_data: String = read\_input("Input other's auth data: ");

    // Copy some values (these are usually the outcome of an X3DH key exchange)

    let bobs\_prekey = KeyPair::new(&mut rng);

    let bobs\_public\_prekey = bobs\_prekey.public().clone();

    let shared = SymmetricKey(

        GenericArray::<u8, U32>::clone\_from\_slice(b"Output of a X3DH key exchange...")

    );

    let mut buff: Vec<u8> = vec![];

    // send hex encoded public key

    prepare\_buff\_to\_send\_msg(

        &mut buff,

        &BASE64\_STANDARD.encode(&our\_auth\_data),

        &BASE64\_STANDARD.encode(&bob\_auth\_data),

        &BASE64\_STANDARD.encode(&String::from("")),

        &BASE64\_STANDARD.encode(&bobs\_public\_prekey.0.as\_bytes())

    );

    // prepare\_buff\_to\_send\_msg(&mut buff, &our\_auth\_data, &bob\_auth\_data, &String::from("sending init key to Alice......."));

    client.write\_all(&buff).expect("Writing to socket failed");

    thread::spawn(move || {

        loop {

            let mut buff = vec![0; BASE\_MSG\_SIZE];

            // Read message

            match client.read\_exact(&mut buff) {

                Ok(\_) => {

                    // println!("Message received ===============================");

                    // Read full message

                    let total\_len = u32::from\_le\_bytes(buff[0..4].try\_into().unwrap()) as usize;

                    buff = if total\_len > BASE\_MSG\_SIZE {

                        // Read remaining data

                        let remaining = total\_len - BASE\_MSG\_SIZE;

                        let mut remaining\_buff = vec![0; remaining];

                        let \_ = client.read\_exact(&mut remaining\_buff);

                        // Combine buffers

                        let mut full\_buff = buff;

                        full\_buff.extend(remaining\_buff);

                        full\_buff

                    } else {

                        buff

                    };

                    let (sent\_from\_auth\_data\_b64, header\_b64, data\_b64) = read\_msg\_from\_buff(

                        &buff,

                        &BASE64\_STANDARD.encode(&our\_auth\_data),

                        &BASE64\_STANDARD.encode(&bob\_auth\_data)

                    );

                    let sent\_from\_auth\_data = BASE64\_STANDARD.decode(

                        &sent\_from\_auth\_data\_b64

                    ).unwrap();

                    let header = BASE64\_STANDARD.decode(&header\_b64).unwrap();

                    let header\_bytes = header.as\_slice();

                    // fill random 0-bytes to the size of the struct to init

                    use ksi\_double\_ratchet::Header;

                    let mut header: Header<PublicKey> = bytes\_to\_type::<Header<PublicKey>>(

                        [0; 40].as\_ref()

                    );

                    if header\_bytes.len() > 0 {

                        header = bytes\_to\_type::<Header<PublicKey>>(header\_bytes);

                    } else {

                    }

                    let data = BASE64\_STANDARD.decode(&data\_b64).unwrap();

                    if sent\_from\_auth\_data == "".as\_bytes() || data == "".as\_bytes() {

                        continue;

                    }

                    if !session\_created {

                        // if string msg\_text starts with [⚙️], then we are Bob and session is created

                        // if data.starts\_with("[⚙️]".as\_bytes()) {

                        if header\_bytes.len() > 0 {

                            bob = SignalDR::new\_bob(shared.clone(), bobs\_prekey.clone(), None);

                            user\_kind = UserKind::Bob;

                            let \_ = bob

                                .ratchet\_decrypt(&header, &data, &bob\_auth\_data.as\_bytes())

                                .unwrap();

                            print!("\r\x1b[K"); // Clear current line

                            println!(

                                "{:?}: {:?}",

                                String::from\_utf8(sent\_from\_auth\_data).unwrap(),

                                "[🤙] Session created, I am <Alice> now"

                            );

                            print!("> "); // Reprint prompt

                            io::stdout().flush().expect("Failed to flush stdout");

                            // send msg to alice with bob's public key

                            session\_created = true;

                            continue;

                        }

                        let bob\_public\_key\_bytes = &data.clone();

                        let bob\_public\_key = PublicKey(

                            x25519\_dalek::PublicKey::from(

                                <[u8; 32]>::try\_from(bob\_public\_key\_bytes.as\_slice()).unwrap()

                            )

                        );

                        // Alice fetches Bob's prekey bundle and completes her side of the X3DH handshake

                        alice = SignalDR::new\_alice(

                            &shared.clone(),

                            bob\_public\_key,

                            None,

                            &mut rng

                        );

                        user\_kind = UserKind::Alice;

                        print!("\r\x1b[K"); // Clear current line

                        println!(

                            "{:?}: {:?}",

                            String::from\_utf8(sent\_from\_auth\_data).unwrap(),

                            "[🤙] Session created, I am <Bob> now"

                        );

                        print!("> "); // Reprint prompt

                        io::stdout().flush().expect("Failed to flush stdout");

                        // send msg to bob with first encrypted message

                        let plain = "Hello Bob, this is Alice".as\_bytes();

                        let (h, ct) = alice.ratchet\_encrypt(

                            &plain,

                            &our\_auth\_data.as\_bytes(),

                            &mut rng

                        );

                        let h\_u8: &[u8] = unsafe {

                            std::slice::from\_raw\_parts(

                                &h as \*const \_ as \*const u8,

                                std::mem::size\_of\_val(&h)

                            )

                        };

                        // println!("hALICE len: {:?}", h\_u8.len());

                        let mut buff: Vec<u8> = vec![];

                        prepare\_buff\_to\_send\_msg(

                            &mut buff,

                            &BASE64\_STANDARD.encode(&our\_auth\_data),

                            &BASE64\_STANDARD.encode(&bob\_auth\_data),

                            &BASE64\_STANDARD.encode(&h\_u8),

                            &BASE64\_STANDARD.encode(&ct)

                        );

                        client.write\_all(&buff).expect("Writing to socket failed");

                        // prepare\_buff\_to\_send\_msg(

                        //     &mut buff,

                        //     &BASE64\_STANDARD.encode(&our\_auth\_data),

                        //     &BASE64\_STANDARD.encode(&bob\_auth\_data),

                        //     &BASE64\_STANDARD.encode(&String::from("")),

                        //     &BASE64\_STANDARD.encode(&String::from("[⚙️]"))

                        // );

                        session\_created = true;

                        // println!("[dbg] session\_created: {:?}", session\_created);

                    } else {

                        // println!("[dbg] user\_kind: {:?}", user\_kind);

                        if user\_kind == UserKind::Alice {

                            // println!("[dbg] Begin decrypting message A =====================");

                            let decrypted = alice

                                .ratchet\_decrypt(&header, &data, &bob\_auth\_data.as\_bytes())

                                .unwrap();

                            print!("\r\x1b[K"); // Clear current line

                            println!(

                                "{:?}: {:?}",

                                String::from\_utf8(sent\_from\_auth\_data).unwrap(),

                                String::from\_utf8(decrypted).unwrap()

                            );

                            print!("> "); // Reprint prompt

                            io::stdout().flush().expect("Failed to flush stdout");

                            // println!("[dbg] End decrypting message A =====================");

                        } else if user\_kind == UserKind::Bob {

                            // println!("[dbg] Begin decrypting message B =====================");

                            let decrypted = bob

                                .ratchet\_decrypt(&header, &data, &bob\_auth\_data.as\_bytes())

                                .unwrap();

                            print!("\r\x1b[K"); // Clear current line

                            println!(

                                "{:?}: {:?}",

                                String::from\_utf8(sent\_from\_auth\_data).unwrap(),

                                String::from\_utf8(decrypted).unwrap()

                            );

                            print!("> "); // Reprint prompt

                            io::stdout().flush().expect("Failed to flush stdout");

                            // println!("[dbg] End decrypting message B =====================");

                        } else {

                            println!("NOT ALICE NOR BOB ====================");

                            print!("\r\x1b[K"); // Clear current line

                            println!("{:?}: {:?}", sent\_from\_auth\_data, data);

                            print!("> "); // Reprint prompt

                            io::stdout().flush().expect("Failed to flush stdout");

                        }

                    }

                }

                Err(ref err) if err.kind() == ErrorKind::WouldBlock => (),

                Err(\_) => {

                    println!("Connection with server was severed");

                    break;

                }

            }

            // Send message

            match rx.try\_recv() {

                Ok(msg) => {

                    let mut buff: Vec<u8> = vec![];

                    // println!("[dbg] user\_kind: {:?}", user\_kind);

                    if user\_kind == UserKind::Alice {

                        // encrypt message

                        // println!("[dbg] Begin encrypting message A =====================");

                        let (h, ct) = alice.ratchet\_encrypt(

                            &msg.as\_bytes(),

                            &our\_auth\_data.as\_bytes(),

                            &mut rng

                        );

                        let h\_u8: &[u8] = unsafe {

                            std::slice::from\_raw\_parts(

                                &h as \*const \_ as \*const u8,

                                std::mem::size\_of\_val(&h)

                            )

                        };

                        // println!("hALICE len: {:?}", h\_u8.len());

                        prepare\_buff\_to\_send\_msg(

                            &mut buff,

                            &BASE64\_STANDARD.encode(&our\_auth\_data),

                            &BASE64\_STANDARD.encode(&bob\_auth\_data),

                            &BASE64\_STANDARD.encode(&h\_u8),

                            &BASE64\_STANDARD.encode(&ct)

                        );

                        client.write\_all(&buff).expect("Writing to socket failed");

                        // println!("[dbg] End encrypting message A =====================");

                    } else if user\_kind == UserKind::Bob {

                        // encrypt message

                        // println!("[dbg] Begin encrypting message B =====================");

                        let (h, ct) = bob.ratchet\_encrypt(

                            &msg.as\_bytes(),

                            &our\_auth\_data.as\_bytes(),

                            &mut rng

                        );

                        let h\_u8: &[u8] = unsafe {

                            std::slice::from\_raw\_parts(

                                &h as \*const \_ as \*const u8,

                                std::mem::size\_of\_val(&h)

                            )

                        };

                        // println!("hBOB len: {:?}", h\_u8.len());

                        prepare\_buff\_to\_send\_msg(

                            &mut buff,

                            &BASE64\_STANDARD.encode(&our\_auth\_data),

                            &BASE64\_STANDARD.encode(&bob\_auth\_data),

                            &BASE64\_STANDARD.encode(&h\_u8),

                            &BASE64\_STANDARD.encode(&ct)

                        );

                        client.write\_all(&buff).expect("Writing to socket failed");

                        // println!("[dbg] End encrypting message B =====================");

                    } else {

                        // println!("NOT ALICE NOR BOB ====================");

                        print!("\r\x1b[K"); // Clear current line

                        println!("[ERROR] SESSION IS NOT ESTABLISHED");

                        print!("> "); // Reprint prompt

                        io::stdout().flush().expect("Failed to flush stdout");

                        prepare\_buff\_to\_send\_msg(

                            &mut buff,

                            &BASE64\_STANDARD.encode(&our\_auth\_data),

                            &BASE64\_STANDARD.encode(&bob\_auth\_data),

                            &BASE64\_STANDARD.encode(&String::from("")),

                            &BASE64\_STANDARD.encode(&msg)

                        );

                        client.write\_all(&buff).expect("Writing to socket failed");

                    }

                    // prepare\_buff\_to\_send\_msg(

                    //     &mut buff,

                    //     &our\_auth\_data,

                    //     &bob\_auth\_data,

                    //     &String::from(""),

                    //     &msg

                    // );

                    // client.write\_all(&buff).expect("Writing to socket failed");

                    // println!("Message sent {:?}", msg);

                    // println!("Buff sent {:?}", buff);

                }

                Err(TryRecvError::Empty) => (),

                Err(TryRecvError::Disconnected) => {

                    break;

                }

            }

            thread::sleep(Duration::from\_millis(100));

        }

    });

    println!("Write a Message:");

    loop {

        // let mut buff = String::new();

        // io::stdin()

        //     .read\_line(&mut buff)

        //     .expect("Reading from stdin failed");

        let buff = read\_input("> ");

        let msg = buff.trim().to\_string();

        if msg == ":quit" || tx.send(msg).is\_err() {

            break;

        }

    }

    println!("bye bye!");

}

// To run this program you need to open 2 terminals. One for the client and one for the server.

// In the server run `cargo run`.

// Then do the same in your client. And this time you should see a message, `write a message`.

// Type something and then you should see that in the server.

// If you type ':quit' then the program will quit...