Лекция 9: Симметричное шифрование

Тема: Криптография - симметричная и асимметричная

Технический университет Молдовы

Лектор: Максим Масютин

Введение

Здравствуйте. Добро пожаловать на Лекцию 9, в которой мы подробно рассмотрим увлекательный мир симметричного шифрования. Это, пожалуй, одна из наиболее фундаментальных тем информационной безопасности, и её глубокое понимание послужит основой для всего остального, что мы будем обсуждать в данном курсе.

Позвольте начать с простого вопроса: что происходит, когда вы отправляете сообщение через интернет? Как ваш банк гарантирует, чтобы никто не мог прочитать баланс вашего счёта, пока он проходит через десятки маршрутизаторов и сетей? Ответ в большинстве случаев связан с симметричным шифрованием.

Задумайтесь: прямо сейчас, в эту минуту, по всему миру активны миллиарды зашифрованных соединений. Каждый HTTPS-сайт, каждое приложение для защищённого обмена сообщениями, каждое VPN-соединение использует симметричное шифрование для основного объёма шифрования данных. Хотя асимметричное шифрование мы будем изучать на следующей неделе, я хочу, чтобы вы поняли: именно симметричное шифрование выполняет основную работу в современных системах безопасности.

Сегодня мы рассмотрим принципы, обеспечивающие работу симметричного шифрования, исследуем как исторические, так и современные алгоритмы, а также разберёмся, как правильно применять эти инструменты в реальных приложениях. К концу лекции вы будете понимать не только то, что делают эти алгоритмы, но и почему они работают и как выбрать подходящий алгоритм для ваших конкретных задач.

Позвольте обозначить план сегодняшней лекции. Сначала мы рассмотрим фундаментальные принципы — математические и концептуальные основы, общие для всех симметричных шифров. Затем сравним блочные шифры и потоковые шифры, чтобы понять, когда использовать каждый из них. Мы совершим исторический экскурс по DES и Тройному DES, чтобы понять, почему они более не пригодны. Основная часть нашего времени будет посвящена AES —

действующему стандарту, включая его различные режимы работы. Мы также рассмотрим ChaCha20 — современного конкурента AES, и Ascon — новый стандарт облегчённой криптографии. Наконец, мы обсудим управление ключами — зачастую самое слабое звено криптографических систем.

Часть 1: Принципы симметричного шифрования

Базовая концепция

Симметричное шифрование, также называемое криптографией с секретным ключом, является старейшей и наиболее интуитивно понятной формой шифрования. Фундаментальная идея удивительно проста: и отправитель, и получатель используют один и тот же секретный ключ. Этот ключ применяется как для шифрования открытого текста в шифротекст, так и для расшифрования шифротекста обратно в открытый текст.

Представьте это как физический ящик с замком. У вас есть ящик с замком, и вы изготавливаете две копии одного ключа. Одну оставляете себе, другую отдаёте другу. Теперь вы можете обмениваться секретными сообщениями, помещая их в ящик. Любой, кто перехватит ящик, не сможет прочитать сообщение без ключа. Эта аналогия использовалась на протяжении веков, и базовый принцип остаётся неизменным в современной криптографии.

Математически это выражается следующим образом:

- Шифрование: C = E(K, P), где C — шифротекст, K — ключ, а P — открытый текст
- Расшифрование: P = D(K, C), где мы восстанавливаем открытый текст с помощью того же ключа

Функция шифрования E принимает два входных параметра — ключ и открытый текст — и производит шифротекст. Функция расшифрования D выполняет обратный процесс с использованием того же ключа. Именно эта симметрия в использовании ключа дала симметричному шифрованию его название.

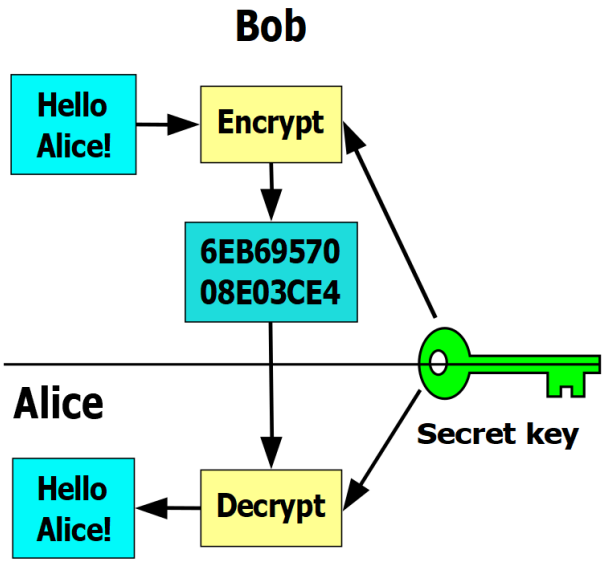


Рисунок 1: Симметричное шифрование: один ключ для шифрования и дешифрования

Безопасность всей системы зависит от сохранения ключа в тайне. Если злоумышленник завладеет ключом, вся защита будет утрачена. Это принципиально отличается от асимметричного шифрования, которое мы будем изучать на следующей неделе, где для шифрования и расшифрования используются разные ключи.

Позвольте подчеркнуть нечто критически важное: ключ — это всё. Вы можете использовать самый изощрённый алгоритм шифрования из когда-либо разработанных, но если ваш ключ слаб, предсказуем или скомпрометирован, ваша защита ничего не стоит. Мы будем неоднократно возвращаться к этой теме на протяжении лекции.

Исторический контекст: от Цезаря до компьютера

Прежде чем перейти к современным алгоритмам, позвольте кратко упомянуть историческое развитие симметричного шифрования. Шифр Цезаря, использовавшийся самим Юлием Цезарем, просто сдвигал каждую букву на фиксированное значение. Если ваш ключ был 3, A становилась D, B становилась E и так далее. Это симметричный шифр — одно и то же значение сдвига шифрует и расшифровывает.

Слабость шифра Цезаря очевидна: существует всего 26 возможных ключей. Вы можете перебрать их все за считанные минуты вручную. Это подводит нас к важному принципу: пространство ключей должно быть достаточно большим, чтобы полный перебор был практически невозможен.

На протяжении веков шифры становились сложнее. Шифр Виженера использовал ключевое слово для определения переменных сдвигов. Машина Энигма использовала роторы и коммутационные панели для создания сложных

подстановок. Каждое усовершенствование увеличивало пространство ключей и сложность. Но каждый из этих шифров был в конечном итоге вскрыт, нередко с разрушительными последствиями в военное время. Криптографические провалы Энигмы мы детально проанализируем в разделе исторических примеров ниже.

Урок из этой истории ясен: криптографическая защита не вечна. Алгоритмы, которые сегодня кажутся невзламываемыми, могут пасть перед достижениями в математике или вычислительной мощности. Именно поэтому мы постоянно оцениваем и обновляем наши криптографические стандарты.

Принципы Шеннона: перемешивание и рассеивание

В 1949 году Клод Шеннон, основатель теории информации, опубликовал основополагающую работу под названием "Communication Theory of Secrecy Systems" ("Теория связи в секретных системах"). Эта работа, наряду с его более ранними исследованиями по теории информации, заложила математические основы криптографии. В ней он ввёл два принципа, которые по сей день являются фундаментом проектирования современных шифров: перемешивание и рассеивание.

Перемешивание означает создание максимально сложной связи между шифротекстом и ключом. Если вы измените один бит ключа, шифротекст должен измениться непредсказуемым образом. Хороший шифр должен делать невозможным определение какой-либо информации о ключе путём анализа шифротекста, даже при наличии пар открытого текста и шифротекста. Это обычно достигается с помощью операций подстановки, при которых байты или биты заменяются согласно сложным правилам, определяемым ключом.

Приведу конкретный пример перемешивания. В AES операция SubBytes заменяет каждый байт с помощью таблицы подстановки, называемой S-блоком. Этот S-блок тщательно спроектирован так, чтобы небольшие изменения входных данных вызывали большие изменения в выходных. Связь между входом и выходом является высоконелинейной, что означает невозможность предсказать выход по входу с помощью простых математических зависимостей.

Рассеивание означает распространение влияния каждого бита открытого текста на множество битов шифротекста. Если вы изменяете один бит открытого текста, должно измениться не менее половины битов шифротекста. Аналогично, изменение одного бита шифротекста должно затронуть множество битов восстановленного открытого текста. Это не позволяет злоумышленникам анализировать закономерности в открытом тексте по шифротексту. Рассеивание обычно достигается с помощью операций перестановки и перемешивания.

Рассмотрим, что происходит без рассеивания. Если бы изменение одного бита открытого текста изменяло лишь один бит шифротекста, злоумышленник мог бы анализировать шифротекст побитно, что радикально сокращает пространство

поиска. При надлежащем рассеивании изменение любого бита вызывает каскад изменений по всему шифротексту, делая такой анализ невозможным.

Современные шифры достигают и перемешивания, и рассеивания через множество раундов операций. Каждый раунд применяет подстановку (для перемешивания) и перестановку (для рассеивания), и эффекты усиливаются с каждым раундом. Именно поэтому AES использует 10, 12 или 14 раундов в зависимости от размера ключа — меньшее количество раундов оставило бы статистические закономерности, которые злоумышленники могли бы использовать.

Точное количество раундов определяется путём криптоанализа. Разработчики анализируют, сколько раундов необходимо для достижения полного рассеивания и сколько раундов обеспечивают достаточный запас прочности против известных методов атак. Затем они добавляют дополнительные раунды для безопасности. Вот почему уменьшение числа раундов в реализации шифра, даже незначительное, может катастрофически ослабить защиту.

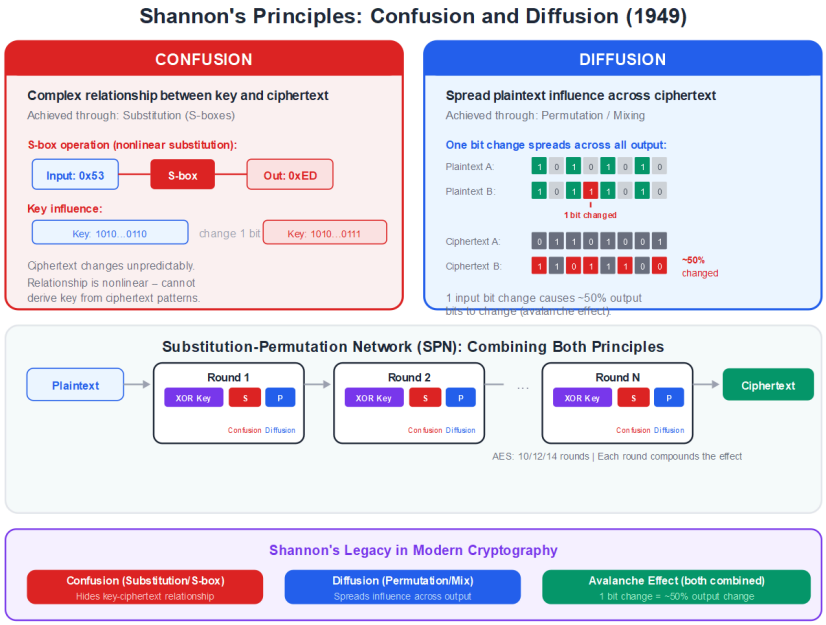


Рисунок 2: Принципы Шеннона: перемешивание и рассеивание

Лавинный эффект

С рассеиванием тесно связан лавинный эффект. Хороший шифр должен демонстрировать выраженное лавинное поведение: изменение одного бита входных данных (будь то открытый текст или ключ) должно менять приблизительно 50% выходных битов, причём каждый выходной бит должен иметь 50%-ную вероятность изменения.

Продемонстрирую это мысленным экспериментом. Предположим, мы шифруем сообщение "Hello" алгоритмом AES с определённым ключом и получаем некоторый шифротекст. Теперь изменим один бит в ключе. Если шифр обладает

хорошими лавинными свойствами, новый шифротекст будет отличаться от исходного приблизительно на 64 из 128 бит. Более того, не будет никакой закономерности в том, какие биты изменились — это будет выглядеть случайно.

Лавинный эффект важен, поскольку он не позволяет злоумышленникам получать информацию путём инкрементного анализа. Если бы изменение одного бита ключа изменяло лишь несколько битов шифротекста, злоумышленник мог бы потенциально искать ключ побитно, что потребовало бы лишь n*2 попыток вместо 2^n.

Принцип Керкгоффса

Прежде чем продолжить, позвольте подчеркнуть важнейший принцип, сформулированный Огюстом Керкгоффсом, нидерландским криптографом, в 1883 году: криптографическая система должна быть безопасной, даже если всё о системе, за исключением ключа, является общедоступным.

На первый взгляд это может показаться нелогичным. Разве не следует держать в секрете и алгоритм тоже? Ответ — нет, и вот почему. История неоднократно показывала, что секретные алгоритмы в конечном счёте подвергаются обратной разработке, утечке или обнаружению. Когда это происходит, все системы, использующие этот алгоритм, немедленно оказываются скомпрометированными. Более того, секретные алгоритмы не могут быть публично проанализированы криптографами, поэтому слабости могут оставаться незамеченными годами.

Рассмотрим пример шифрования CSS, использовавшегося в DVD. Алгоритм хранился в тайне, но в конечном итоге был подвергнут обратной разработке в 1999 году. Обнаруженные слабости — эффективная длина ключа в 40 бит и ошибочная процедура формирования ключей — присутствовали всё время, но были скрыты от публичного анализа. Если бы алгоритм был публичным, эти слабости были бы обнаружены и устранены значительно раньше.

Современные криптографические алгоритмы, такие как AES и ChaCha20, полностью открыты. Любой может скачать спецификации и реализовать их. Их безопасность полностью основана на секретности ключа, а не на скрытности алгоритма. Этот подход подтверждён десятилетиями публичного криптоанализа — эти алгоритмы были изучены тысячами исследователей и остаются безопасными.

Этот принцип имеет практическое значение для вас как специалистов по безопасности: никогда не доверяйте "проприетарным" алгоритмам шифрования. Если вендор не сообщает, какой алгоритм он использует, считайте его слабым. Требуйте стандартные алгоритмы, такие как AES или ChaCha20.

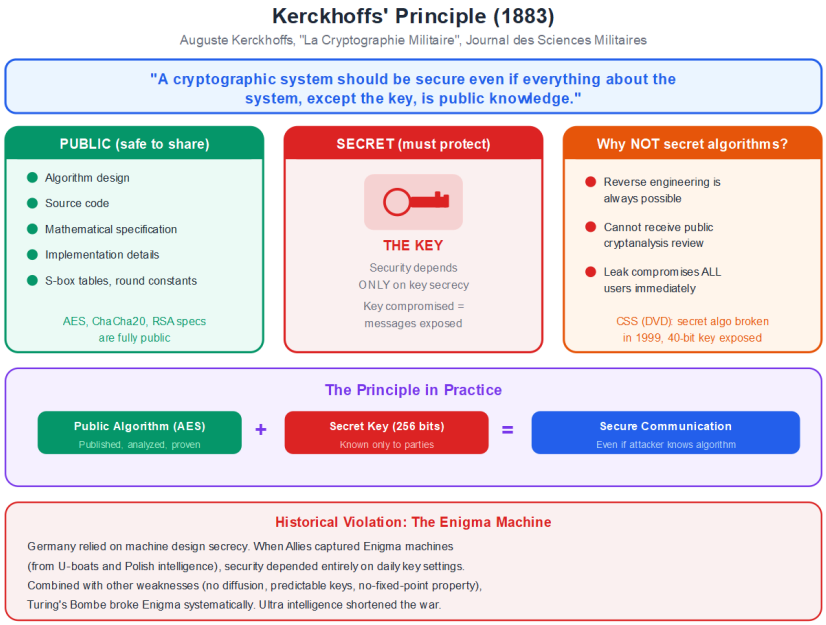


Рисунок 3: Принцип Керкгоффса: безопасность зависит от ключа, а не от алгоритма

Вычислительная безопасность

Ещё одна базовая концепция, прежде чем мы двинемся дальше: современная криптография основана на вычислительной безопасности, а не на теоретико-информационной безопасности.

Теоретико-информационная безопасность означает, что шифротекст не содержит никакой информации об открытом тексте, независимо от вычислительной мощности. Одноразовый блокнот обеспечивает именно это — если вы выполняете XOR вашего сообщения с действительно случайным ключом равной длины, шифротекст абсолютно защищён. Однако это требует ключей такой же длины, как и сообщения, что непрактично для большинства приложений.

Вычислительная безопасность означает, что шифротекст теоретически может содержать информацию об открытом тексте, но для её извлечения потребуются вычислительные ресурсы, превышающие любые практические пределы. Когда мы говорим, что AES-256 предоставляет 256-битную защиту, мы имеем в виду, что самая быстрая известная атака требует приблизительно 2^256 операций. Даже если бы у вас был компьютер, выполняющий триллион операций в секунду, работающий со времён Большого взрыва, вы бы не завершили даже крошечную долю этого поиска.

Это основа всей практической симметричной криптографии: мы проектируем алгоритмы, где единственной осуществимой атакой является полный перебор ключей, и делаем ключи достаточно длинными, чтобы перебор был невозможен.

Одноразовый блокнот и совершенная секретность

Одноразовый блокнот (OTP) — единственная схема шифрования, математически доказанно невзламываемая при корректном использовании. В одноразовом блокноте ключ представляет собой случайную последовательность, не короче самого сообщения, каждый ключ используется ровно один раз, а операция шифрования — простое XOR (или модулярное сложение) открытого текста с ключом.

В 1949 году Клод Шеннон опубликовал работу "Communication Theory of Secrecy Systems" в Bell System Technical Journal. В этой основополагающей статье Шеннон доказал, что одноразовый блокнот обеспечивает то, что он назвал **совершенной секретностью**: шифротекст не раскрывает абсолютно никакой информации об открытом тексте для злоумышленника, независимо от его вычислительной мощности. Точнее, для любого шифротекста все возможные открытые тексты той же длины равновероятны.

Шеннон также доказал важный отрицательный результат: **любой шифр, обеспечивающий совершенную секретность, должен иметь не меньше возможных ключей, чем возможных сообщений**. На практике это означает, что ключ должен быть не короче сообщения. Таким образом, одноразовый блокнот является не просто одним из способов достижения совершенной секретности; он, по существу, единственный.

Условия совершенной секретности (все должны выполняться одновременно):

1. **Ключ должен быть истинно случайным** (не псевдослучайным). Любая закономерность или предсказуемость в генерации ключа допускает статистические атаки.
2. **Ключ должен быть не короче открытого текста**. Более короткие ключи повторно используют ключевой материал, создавая закономерности.
3. **Ключ никогда не должен использоваться повторно**. Повторное использование ключа даже единожды позволяет злоумышленнику выполнить XOR двух шифротекстов, устранив ключ и выявив соотношения между открытыми текстами. Это называется атакой "повторного использования ключа" или атакой "двукратного блокнота".
4. **Ключ должен храниться в полной тайне**. Если ключ скомпрометирован, все зашифрованные им сообщения раскрыты.

Почему одноразовый блокнот непрактичен: эти условия крайне трудно выполнить на практике. Ключ должен быть такой же длины, как все сообщения вместе взятые, что требует безопасной генерации, распределения, хранения и уничтожения огромных объёмов ключевого материала. Безопасное распределение ключей само по себе является серьёзной проблемой; если бы у вас был защищённый канал для передачи ключа, вы могли бы использовать этот канал для передачи самого сообщения. Именно поэтому практическая криптография

опирается на вычислительную безопасность (AES, ChaCha20), а не на теоретико-информационную.

Исторические примеры

Теперь, когда мы установили теоретические принципы (перемешивание, рассеивание, лавинный эффект, принцип Керкгоффса, совершенная секретность), рассмотрим два исторических случая, иллюстрирующих, как нарушение этих принципов приводит к криптографическому провалу. Эти случаи демонстрируют два принципиально различных типа отказа: дефекты проектирования алгоритма и эксплуатационные ошибки.

Машина Энигма

Машина Энигма, использовавшаяся нацистской Германией во Второй мировой войне, является одним из важнейших примеров в истории криптографии. Энигма представляла собой электромеханическую роторную шифровальную машину, создававшую полиалфавитные подстановочные шифры огромной сложности. Германское военное командование считало Энигму невзламываемой.

Энигма была вскрыта, потому что нарушала ряд принципов, которые мы только что обсудили. Анализ этих нарушений показывает, почему современные шифры проектируются именно так, а не иначе.

Нарушение требования отсутствия неподвижных точек (слабая перестановка): рефлектор Энигмы (Umkehrwalze) направлял электрический сигнал обратно через роторы по другому проводу, что делало физически невозможным шифрование любой буквы в саму себя. Нажатие 'A' никогда не могло дать 'A' на выходе. Надёжный шифр не должен иметь структурных ограничений, сокращающих пространство выходных значений. Это свойство "отсутствия неподвижных точек" означало, что криптоаналитики могли мгновенно исключить любой вариант расшифрования, где буква открытого текста совпадала с соответствующей буквой шифротекста, сокращая пространство поиска на множитель, возрастающий с каждой проверяемой буквой.

Нарушение принципа Керкгоффса: Германия в значительной мере полагалась на секретность самой конструкции машины, и когда союзная разведка получила экземпляры Энигмы (с захваченных подводных лодок и от польской разведки), безопасность системы целиком зависела от ежедневных настроек ключа. Конструкция машины не обеспечивала защиты после её раскрытия.

Недостаточное рассеивание: Энигма работала как подстановочный шифр: каждая буква независимо заменялась другой буквой. Изменение одной буквы открытого текста изменяло только соответствующую букву шифротекста. Рассеивание отсутствовало: не было механизма, благодаря которому изменение в одной части открытого текста влияло бы на другие части шифротекста. У Энигмы не было лавинного эффекта вообще.

Ограниченное перемешивание: подстановка в Энигме определялась позициями роторов, которые продвигались предсказуемо (подобно одометру). Связь между ключом и шифротекстом, хотя и сложная, следовала механической закономерности, которую можно было математически смоделировать. Современные шифры достигают перемешивания через нелинейные операции (S-блоки), делающие связь между ключом и шифротекстом максимально сложной.

Процедурный провал (закономерность повторного использования ключа): немецкие операторы передавали ключ сообщения (начальные позиции роторов) дважды в начале каждого сообщения. Это повторение создавало математические закономерности. Польский математик Мариан Реевский, криптоаналитик, использовал это в 1932 году, смоделировав роторы Энигмы как группы перестановок и решив полученную систему уравнений. Этот подход свёл задачу от полного перебора всего пространства ключей к алгебраическому анализу закономерности повторяющегося ключа. Повторение ключа нарушало базовый принцип: никогда не создавать предсказуемую структуру в шифротексте.

Слабое управление ключами: многие операторы выбирали предсказуемые ключи сообщений (например, "AAA" или инициалы своих подруг), ещё больше ослабляя систему. Предсказуемые ключи сокращают эффективное пространство ключей от теоретического максимума до малого подмножества, которое злоумышленник может перечислить.

Когда Германия усовершенствовала процедуры Энигмы, Алан Тьюринг, британский математик и криптоаналитик, и его коллеги в Блетчли-Парке разработали Бомбу — электромеханическое устройство, систематически проверявшее настройки Энигмы с использованием свойства отсутствия неподвижных точек и известного открытого текста (подсказок, таких как сводки погоды с предсказуемыми фразами). К 1940 году британская разведка регулярно дешифровала трафик Энигмы (операция с кодовым названием Ultra).

Случай с Энигмой иллюстрирует, что криптосистема надёжна ровно настолько, насколько надёжно её слабейшее звено. Энигма нарушала принцип Керкгоффса, не обладала рассеиванием и лавинным эффектом, имела структурный дефект (отсутствие неподвижных точек) и страдала от неудовлетворительных процедур управления ключами. Современные симметричные шифры, такие как AES, специально спроектированы для удовлетворения всех этих свойств.

Проект ВЕНОНА и последствия повторного использования ключей

Проект ВЕНОНА — хорошо задокументированный пример того, что происходит при нарушении правила одноразового блокнота "никогда не использовать ключ повторно".

Во время Второй мировой войны Советский Союз использовал одноразовые блокноты для наиболее секретных дипломатических и разведывательных

коммуникаций. При правильном использовании эти сообщения были теоретически невзламываемыми. Однако во время наступления немцев на Москву в конце 1941 и начале 1942 года советская фабрика, изготавливавшая книги одноразовых ключей, оказалась под сильнейшим давлением, вынуждавшим производить больше ключевого материала, чем она могла надлежащим образом сгенерировать. Чтобы удовлетворить спрос, фабрика изготовила приблизительно 35 000 страниц дублированного ключевого материала. Эти дубликаты были разосланы в разные, географически удалённые советские учреждения в надежде, что повторное использование никогда не будет обнаружено.

Фундаментальное правило, которое было нарушено: **один и тот же ключевой материал использовался для шифрования более одного сообщения**. Когда два сообщения зашифрованы одним и тем же ключом одноразового блокнота, злоумышленник может выполнить XOR двух шифротекстов. Ключ взаимно уничтожается, оставляя XOR двух открытых текстов. Используя известные закономерности открытого текста, распространённые фразы и статистический анализ (метод, называемый "протаскивание подсказки"), криптоаналитики могут постепенно восстановить оба сообщения.

Начиная с 1943 года американские криптоаналитики в Арлингтон-Холле начали собирать советский дипломатический трафик. В 1946 году Мередит Гарднер, американский криптоаналитик, выявил первые признаки повторного использования ключей. В последующие годы команда ВЕНОНА (совместное американо-британское подразделение) частично дешифровала приблизительно 3000 советских сообщений из сотен тысяч перехваченных.

Дешифрованные сообщения раскрыли советские шпионские сети в Соединённых Штатах и Великобритании, включая агентов в Манхэттенском проекте (программе атомной бомбы), Государственном департаменте и британской разведке. Среди выявленных через ВЕНОНА шпионов были Клаус Фукс, Юлиус Розенберг, Дональд Маклин и Ким Филби.

Принципы, нарушенные в случае ВЕНОНЫ:

Нарушение требования уникальности ключа (условие 3 Шеннона): Шеннон доказал, что совершенная секретность требует, чтобы каждый ключ использовался ровно один раз. Советская фабрика произвела дублированные страницы ключей, напрямую нарушив это условие. Как только два шифротекста используют общий ключ, их XOR устраняет ключ и обнажает соотношение между двумя открытыми текстами. Математически это эквивалентно шифрованию ключом из одних нулей: никакой защиты не остаётся.

Нарушение требования к длине ключа (частичное): некоторые советские учреждения под давлением также повторно использовали фрагменты ключевых страниц или применяли ключевой материал не в том порядке, ещё больше снижая защиту. Любой фрагмент ключевого материала, использованный более одного раза, создаёт окно для криптоанализа в данной позиции.

Нарушение процедур управления ключами: решение дублировать книги ключей было эксплуатационным провалом, а не алгоритмическим. Алгоритм одноразового блокнота так и не был вскрыт. Фабрика подменила удобством (копирование ключевых страниц) строгую случайную генерацию, которую требовала система. Это параллель со случаем Энигмы: слабым звеном был не алгоритм, а процедуры его использования.

Перемешивание и рассеивание не были нарушены: в отличие от Энигмы, одноразовый блокнот по своей природе обеспечивает и совершенное перемешивание (каждый бит шифротекста зависит от уникального случайного бита ключа), и совершенное рассеивание (для восстановления любого открытого текста необходим весь ключ). Случай ВЕНОНЫ не использовал слабости в конструкции шифра. Он использовал единственный эксплуатационный провал: повторное использование ключа.

Случай ВЕНОНЫ доказывает теорему Шеннона в конкретных терминах: одноразовый блокнот невзламываем только при выполнении всех четырёх условий. Нарушение даже одного условия (уникальности ключа) превратило шифр с совершенной секретностью в шифр, поддающийся стандартным криптоаналитическим методам.

Энигма и одноразовый блокнот: два типа уязвимости

Сравнение случаев Энигмы и ВЕНОНЫ выявляет два принципиально различных способа отказа криптосистемы. Энигма имела дефекты проектирования: ей не хватало рассеивания и лавинного эффекта, перемешивание было слабым из-за механической структуры, она нарушала принцип Керкгоффса, полагаясь на секретность машины, и имела структурный дефект — отсутствие неподвижных точек. Это дефекты самого алгоритма. Одноразовый блокнот, напротив, не имел дефектов проектирования: он обеспечивает математически доказанную совершенную секретность. Он был вскрыт исключительно через эксплуатационный провал: фабрика дублировала ключевой материал, нарушив условие уникальности ключа. Урок для современной криптографии: и алгоритм, и его эксплуатационная среда должны быть безопасны. AES решает алгоритмическую сторону (сильные перемешивание, рассеивание и лавинный эффект, полное соответствие принципу Керкгоффса). Практики управления ключами, обращение с одноразовыми номерами и безопасная генерация ключей решают эксплуатационную сторону.

Часть 2: Блочные шифры и потоковые шифры

Симметричные шифры существуют в двух фундаментальных разновидностях: блочные шифры и потоковые шифры. Понимание различия между ними необходимо для выбора правильного инструмента для каждой ситуации.

Блочные шифры

Блочный шифр оперирует блоками данных фиксированного размера. Например, AES обрабатывает данные 128-битными (16-байтовыми) блоками. Шифр принимает блок открытого текста и ключ и выдаёт блок шифротекста того же размера. Операция является детерминированной — один и тот же блок открытого текста и ключ всегда производят один и тот же блок шифротекста.

Если ваш открытый текст составляет ровно 16 байт, прекрасно — вы шифруете его напрямую. Но что если ваш открытый текст длиннее или короче?

Для открытого текста длиннее одного блока используются режимы работы, которые мы подробно обсудим далее. Режимы работы определяют, как безопасно обрабатывать несколько блоков, включая способы добавления рандомизации, чтобы идентичные блоки открытого текста не порождали идентичные блоки шифротекста.

Для открытого текста короче одного блока необходимо дополнение — добавление дополнительных байтов для заполнения блока. Наиболее распространённая схема дополнения — PKCS#7, при которой добавляются байты, значение каждого из которых равно количеству необходимых байтов дополнения. Например, если нужно 5 байтов дополнения, добавляются пять байтов, каждый со значением 0x05. Если нужен 1 байт дополнения, добавляется 0x01. Если ваше сообщение точно кратно размеру блока, добавляется полный блок 0x10 (16 байтов, каждый содержит 16).

Эта схема дополнения однозначна — получатель всегда может определить, сколько байтов дополнения нужно удалить, посмотрев на значение последнего байта. Однако дополнение было источником уязвимостей, в частности атак на оракул дополнения, которые мы обсудим позже.

Блочные шифры сами по себе детерминированы — один и тот же открытый текст и ключ всегда производят один и тот же шифротекст. Это на самом деле является проблемой, поэтому они всегда используются с режимами работы, которые вносят случайность через векторы инициализации. Мы рассмотрим это подробно при обсуждении режима ECB и причин его опасности.

Внутренняя структура блочных шифров обычно включает множество раундов операций. Каждый раунд применяет некоторую комбинацию подстановки,

перестановки и подмешивания ключа. Распространённые структуры включают:

- **Сети Фейстеля** (используются в DES): блок разделяется пополам, одна половина обрабатывается раундовой функцией, результат XOR-ится с другой половиной, половины меняются местами, процесс повторяется
- **Подстановочно-перестановочные сети** (используются в AES): подстановка применяется ко всему состоянию, затем перестановка, затем подмешивание ключа, процесс повторяется

Обе структуры способны обеспечить надёжную защиту при достаточном числе раундов и хорошо спроектированных раундовых функциях.

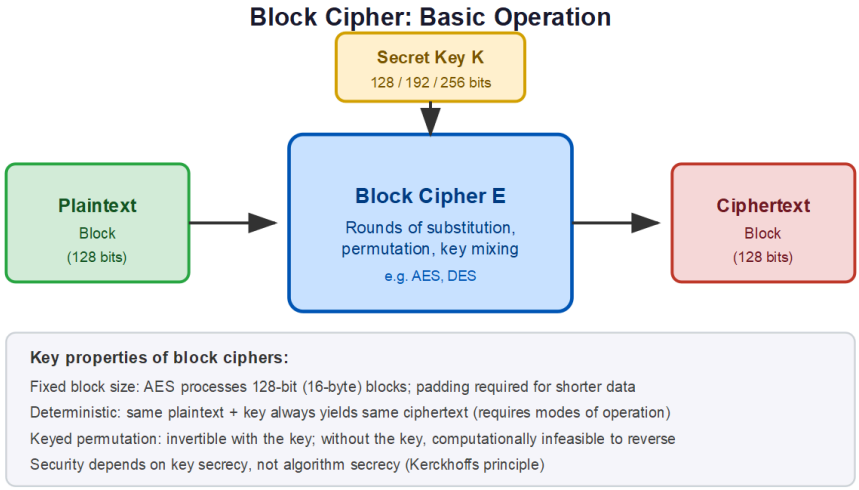


Рисунок 26: Работа блочного шифра: блок открытого текста и ключ дают блок шифротекста того же размера

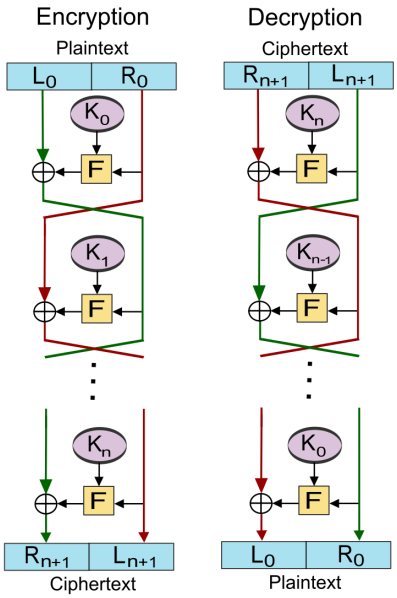


Рисунок 5: Сеть Фейстеля: итеративная структура блочного шифра

Потоковые шифры

Потоковый шифр генерирует псевдослучайную ключевую последовательность (гамму) из ключа и одноразового номера (nonce — числа, используемого однократно). Эта гамма затем XOR-ится с открытым текстом для получения шифротекста. Расшифрование работает идентично — XOR шифротекста с той же гаммой восстанавливает открытый текст.

Математически:

- C = P XOR keystream(K, nonce)
- P = C XOR keystream(K, nonce)

Потоковые шифры концептуально проще и зачастую быстрее блочных шифров. Они могут обрабатывать данные побайтно или даже побитно, что делает их идеальными для потоковых приложений, где данные поступают непрерывно и нет возможности ждать заполнения полного блока перед шифрованием.

Генератор гаммы должен быть криптографически стойким. Должно быть вычислительно невозможно:

- Предсказать будущую гамму по прошлой
- Восстановить ключ по наблюдаемой гамме
- Отличить гамму от истинно случайных данных

ChaCha20 — важнейший современный потоковый шифр. Он генерирует гамму блоками по 64 байта, которые XOR-ятся с открытым текстом. Генерация гаммы является быстрой и допускает распараллеливание.

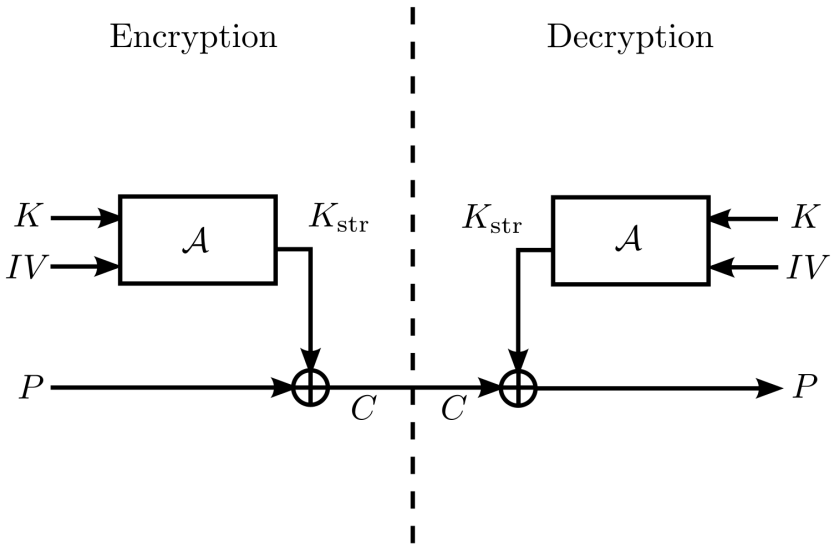


Рисунок 4: Потоковый шифр: шифрование данных побитно или побайтно

Критическое требование для потоковых шифров — никогда не использовать одноразовый номер повторно с тем же ключом. В противном случае злоумышленник может выполнить XOR двух шифротекстов для устранения гаммы, получив XOR двух открытых текстов:

C1 XOR C2 = (P1 XOR keystream) XOR (P2 XOR keystream) = P1 XOR P2

XOR двух открытых текстов часто раскрывает значительную информацию об обоих, особенно если они содержат естественный язык или структурированные данные. Это в точности тот метод, который позволил вскрыть советский одноразовый блокнот в проекте ВЕНОНА: дублированные страницы ключей означали, что разные сообщения были зашифрованы идентичным ключевым материалом, и XOR шифротекстов обнажил соотношения открытых текстов.

Сравнение блочных и потоковых шифров

Аспект	Блочный шифр	Потоковый шифр
Единица обработки	Блоки фиксированного размера	Биты или байты
Требуется дополнение	Да	Нет
Произвольный доступ	Зависит от режима	Да (при возможности позиционирования)
Распараллеливание	Зависит от режима	Обычно да
Распространение ошибок	Зависит от режима	Один бит
Аппаратная поддержка	AES-NI широко распространена	Менее распространена
Основной современный выбор	AES	ChaCha20

На практике граница между ними размыта. Режим CTR превращает блочный шифр в потоковый, а многие потоковые шифры внутренне используют блокоподобные структуры. Выбор между AES и ChaCha20 больше связан с аппаратной поддержкой и свойствами реализации, нежели с разделением на блочные и потоковые.

Часть 3: Устаревшие алгоритмы — DES и Тройной DES

Стандарт шифрования данных (DES)

Начнём наш обзор конкретных алгоритмов с DES — Стандарта шифрования данных. Понимание DES важно не потому, что его следует использовать — категорически не следует — а потому, что он иллюстрирует фундаментальные концепции и объясняет, почему мы располагаем алгоритмами, которые используем сегодня.

DES был разработан компанией IBM в начале 1970-х годов, модифицирован NSA (Агентством национальной безопасности) и принят в качестве федерального стандарта в 1977 году. Более двух десятилетий он был самым распространённым алгоритмом шифрования в мире. Банки, правительства и предприятия по всему миру полагались на DES для защиты конфиденциальной информации.

DES — это блочный шифр, оперирующий 64-битными блоками с 56-битным ключом. Подождите, можете сказать вы — мы обычно говорим о размерах ключей в степенях двойки, так почему 56 бит? Ответ в том, что ключ фактически хранится как 64 бита, но 8 из них являются битами чётности, используемыми для обнаружения ошибок, оставляя только 56 бит реального ключевого материала. Это проектное решение, повлиянное NSA, было спорным в то время и остаётся таковым — многие полагали, что NSA намеренно ослабило алгоритм.

Алгоритм состоит из 16 раундов структуры, называемой сетью Фейстеля. В каждом раунде:

1. 64-битный блок разделяется на две 32-битные половины: левую (L) и правую (R)
2. Правая половина обрабатывается функцией F, которая включает:

- Расширение с 32 до 48 бит
- XOR с раундовым подключом (48 бит)
- Обработку через 8 подстановочных таблиц (S-блоков), редуцирующих обратно до 32 бит
- Перестановку

3. Выход F XOR-ится с левой половиной
4. Половины меняются местами
5. Процесс повторяется со следующим раундовым подключом

S-блоки — ядро безопасности DES. Они обеспечивают нелинейность (перемешивание), которая делает шифр устойчивым к линейному и

дифференциальному криптоанализу. Конструкция этих S-блоков годами хранилась в тайне, порождая спекуляции о скрытых слабостях. Когда дифференциальный криптоанализ был публично открыт в 1990-х годах, анализ показал, что S-блоки DES на самом деле были оптимально устойчивы к этой атаке — что свидетельствует о том, что NSA знало о дифференциальном криптоанализе за десятилетия до академических исследователей.

DES был прорывом для своего времени, но он имеет два фатальных недостатка для современного использования:

1. **Слишком малый размер ключа**: при всего 56 битах существует 2^56 или около 72 квадриллионов возможных ключей. В 1977 году это казалось невообразимо большим числом. К 1998 году Electronic Frontier Foundation построил машину под названием "Deep Crack", способную взломать DES менее чем за 3 дня при стоимости около 250 000 долларов. Сегодня вы можете взломать DES за часы, используя облачные вычисления за несколько сотен долларов, или с помощью специализированного оборудования — за минуты.
2. **Слишком малый размер блока**: 64-битные блоки приводят к проблемам при шифровании больших объёмов данных одним ключом. После приблизительно 2^32 блоков (32 ГБ) атаки дней рождения на определённые режимы становятся практичными, что ведёт к утечке информации.

Проблема размера ключа была очевидна уже в 1977 году. Исследователи настаивали на не менее 80 битах, предпочтительно больше. Ограничение в 56 бит было политическим компромиссом, и история оправдала критиков.

DES никогда не должен использоваться в новых системах. Он включён сюда исключительно для исторического понимания и потому, что вы можете столкнуться с ним в устаревших системах. В таком случае вашим первым приоритетом должна быть миграция на современные алгоритмы.

Тройной DES (3DES)

Когда слабости DES стали очевидны, индустрии потребовалось быстрое решение. Организации вложили значительные средства в аппаратное и программное обеспечение DES, и полная замена была невозможна в ближайшей перспективе. Решением стал Тройной DES, также называемый 3DES или TDEA (Triple Data Encryption Algorithm — алгоритм тройного шифрования данных). Идея проста: применить DES трижды с разными ключами.

Наиболее безопасный вариант использует три независимых ключа (K1, K2, K3) и выполняет:

- Шифротекст = E(K3, D(K2, E(K1, Открытый текст)))

Обратите внимание, что средняя операция — это расшифрование, а не шифрование. Эта схема "шифрование-расшифрование-шифрование" (EDE) была выбрана для обратной совместимости — если задать K1 = K2 = K3, получается

стандартный DES. Это позволяло реализациям 3DES взаимодействовать с устаревшими системами DES при необходимости.

С тремя 56-битными ключами 3DES имеет номинальную криптостойкость 168 бит. Однако из-за атак типа "встреча посередине" реальная защита с тремя ключами ближе к 112 битам. Атака работает путём шифрования с одной стороны и расшифрования с другой с поиском совпадений в середине. Это значительно снижает эффективную защиту.

Двухключевой вариант (где K1 = K3) также распространён, обеспечивая около 80 бит защиты. Этот вариант использует лишь 112 бит ключевого материала и был популярен, когда 168-битные ключи казались избыточными.

3DES решил проблему размера ключа, но создал новые проблемы:

1. **Очень медленный**: 3DES требует три операции DES на блок, что делает его втрое медленнее DES, который и без того был не быстр по современным стандартам. На современном оборудовании 3DES в 50-100 раз медленнее AES.
2. **По-прежнему 64-битные блоки**: проблема атаки дней рождения остаётся. После шифрования около 32 ГБ одним ключом безопасность деградирует. Атака Sweet32 продемонстрировала это на практике в 2016 году.
3. **Неудобное управление ключами**: управление тремя связанными ключами сложнее, чем управление одним ключом.

NIST (National Institute of Standards and Technology, Национальный институт стандартов и технологий) объявил 3DES устаревшим в 2023 году и установил 2030 год как дату, после которой он не должен использоваться даже в устаревших системах. Если вы работаете с системами, всё ещё использующими 3DES, начинайте планирование миграции уже сейчас.

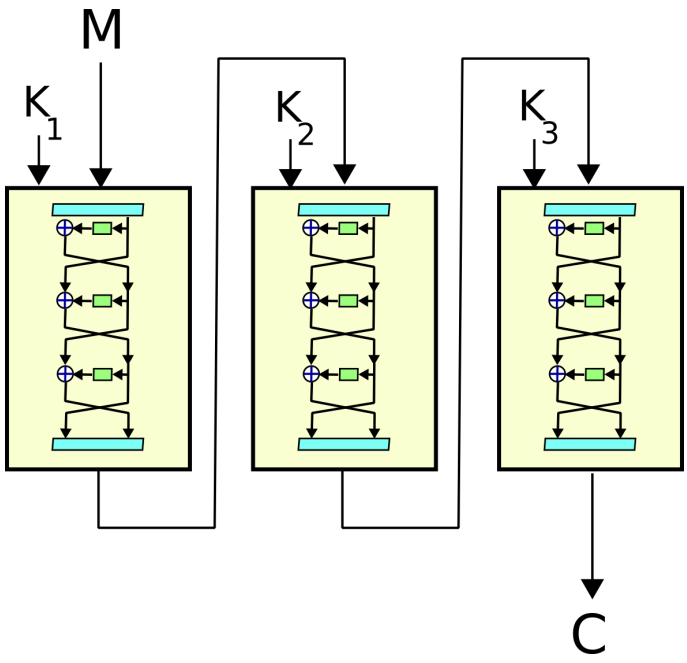


Рисунок 6: Triple DES: три раунда шифрования DES

Урок DES и 3DES ясен: требования к безопасности эволюционируют, и алгоритмы, которые были достаточны десятилетия назад, могут быть опасны сегодня. Всегда используйте актуальные стандарты и планируйте будущую миграцию.

Часть 4: Усовершенствованный стандарт шифрования (AES)

Конкурс AES

К концу 1990-х годов криптографическое сообщество пришло к единому мнению о необходимости замены DES. Вместо разработки алгоритма в тайне (как было с DES), NIST организовал открытый конкурс. Это был революционный подход, который с тех пор стал стандартом для криптографической стандартизации.

Конкурс начался в 1997 году с объявления о приёме заявок. Требования включали:

- 128-битный размер блока (вдвое больше 64 бит DES)
- Поддержку ключей длиной 128, 192 и 256 бит
- Эффективность как в аппаратной, так и в программной реализации
- Отсутствие лицензионных отчислений

Пятнадцать алгоритмов были представлены командами со всего мира. Они прошли три года интенсивного публичного анализа. Криптографы, учёные и практики по всему миру оценивали безопасность, производительность и характеристики реализации. Этот открытый процесс означал, что любые слабости с высокой вероятностью были бы обнаружены до стандартизации.

Пять финалистов:

- MARS (IBM)
- RC6 (RSA Security)
- Rijndael (Жоан Демен и Винсент Рэймен, бельгийские криптографы)
- Serpent (Росс Андерсон, Эли Бихам и Ларс Кнудсен, криптографы-исследователи)
- Twofish (Брюс Шнайер, исследователь в области безопасности, и соавторы)

В 2001 году Rijndael (произносится приблизительно как "рейн-дол") был выбран в качестве нового Усовершенствованного стандарта шифрования. Он победил благодаря сочетанию безопасности, производительности и элегантности. Алгоритм математически чист, эффективно работает на всём — от 8-битных микроконтроллеров до 64-битных серверов, и обладает хорошей устойчивостью к атакам по побочным каналам.

Rijndael был разработан двумя бельгийскими криптографами — Жоаном Деменом и Винсентом Рэйменом. Он основан на их более ранней работе над шифром Square и вводит инновационную "стратегию широкого следа" для обеспечения рассеивания.

Структура AES

AES — блочный шифр, оперирующий 128-битными блоками. В отличие от Rijndael, который поддерживал переменные размеры блоков (128, 192 или 256 бит), стандарт AES фиксировал размер блока на 128 битах. Поддерживаются три размера ключа:

- **AES-128**: 128-битный ключ, 10 раундов
- **AES-192**: 192-битный ключ, 12 раундов
- **AES-256**: 256-битный ключ, 14 раундов

Состояние представляется в виде матрицы 4x4 из байтов (4 строки, 4 столбца, 16 байтов = 128 бит). Каждый раунд преобразует всё состояние с помощью четырёх операций.

Позвольте подробно рассмотреть каждую операцию:

1. SubBytes (подстановка)

Каждый байт состояния заменяется с помощью таблицы подстановки (S-блока). Этот единственный S-блок применяется ко всем 16 байтам независимо. S-блок представляет собой фиксированную таблицу подстановки из 256 элементов, обеспечивающую свойство перемешивания.

S-блок не произволен — он получен из математического обратного элемента в GF(2^8) с последующим аффинным преобразованием. Такая конструкция создаёт высокую нелинейность, то есть выход не может быть аппроксимирован простыми линейными функциями от входа. Эта устойчивость к линейной аппроксимации критически важна для защиты от линейного криптоанализа.

2. ShiftRows (перестановка)

Строки матрицы состояния сдвигаются на различные величины:

- Строка 0: без сдвига
- Строка 1: циклический сдвиг влево на 1 байт
- Строка 2: циклический сдвиг влево на 2 байта
- Строка 3: циклический сдвиг влево на 3 байта

Эта простая операция гарантирует, что байты из каждого столбца распределяются по разным столбцам после следующей операции. Она начинает процесс рассеивания, перемещая данные между столбцами.

3. MixColumns (перемешивание)

Каждый столбец рассматривается как полином и умножается на фиксированный полином в GF(2^8). Более наглядно: каждый выходной байт столбца является определённой линейной комбинацией всех четырёх входных байтов этого столбца.

Эта операция завершает процесс рассеивания. После SubBytes (перемешивание) и ShiftRows, MixColumns обеспечивает, что каждый байт влияет на все четыре байта в своём столбце. В сочетании с ShiftRows это гарантирует, что через несколько раундов каждый байт входных данных влияет на каждый байт выходных.

4. AddRoundKey (подмешивание ключа)

Раундовый ключ XOR-ится с состоянием. Каждый раунд использует отдельный 128-битный раундовый ключ, полученный из основного ключа через процедуру развёртывания ключа.

Процедура развёртывания ключа расширяет исходный ключ (128, 192 или 256 бит) в 11, 13 или 15 раундовых ключей (10, 12 или 14 раундов плюс начальный ключ). Процедура развёртывания использует SubBytes и раундовые константы, чтобы обеспечить достаточное различие раундовых ключей друг от друга.

Структура раундов:

- Начальный AddRoundKey
- Раунды с 1 по (N-1): SubBytes, ShiftRows, MixColumns, AddRoundKey
- Финальный раунд N: SubBytes, ShiftRows, AddRoundKey (без MixColumns)

В финальном раунде MixColumns опускается, поскольку это добавило бы вычислительные затраты без выигрыша в безопасности — его эффект был бы нейтрализован обратной MixColumns в первом раунде расшифрования.

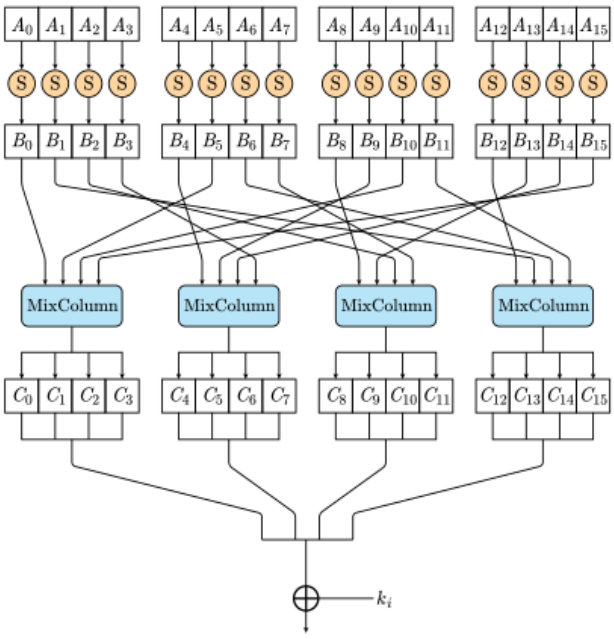


Рисунок 7: Операции раунда AES: SubBytes, ShiftRows, MixColumns, AddRoundKey

Безопасность AES

AES интенсивно изучался более двух десятилетий. Он выдержал обширный криптоанализ лучших криптографов мира. Не обнаружено ни одной практической атаки, превосходящей полный перебор.

Наиболее известные атаки включают:

Атаки типа biclique (2011): они снижают сложность атаки на AES-128 с 2^128 до 2^126.1, на AES-192 с 2^192 до 2^189.7, и на AES-256 с 2^256 до 2^254.4. Хотя это теоретические улучшения, они по-прежнему астрономически невозможны. Разница между 2^256 и 2^254 составляет приблизительно коэффициент 4 — бессмысленный, когда оба числа невообразимо велики.

Атаки на связанных ключах: требуют от злоумышленника шифрования с несколькими ключами, имеющими определённые математические зависимости. Они неприменимы к обычным сценариям использования, где ключи независимы.

Атаки по побочным каналам: атакуют реализации, а не сам алгоритм. Измеряя временные характеристики, потребляемую мощность или электромагнитное излучение, злоумышленники могут получить информацию о ключе. AES несколько уязвим к атакам по времени доступа к кэшу в программных реализациях, использующих таблицы подстановки. Аппаратные реализации с AES-NI в значительной степени защищены.

Несколько важных замечаний о выборе размера ключа AES:

AES-128 обеспечивает 128-битную защиту, что достаточно для большинства приложений сегодня. Этот уровень защиты значительно превышает любые предсказуемые вычислительные возможности. Даже с квантовыми компьютерами алгоритм Гровера лишь снижает это до 64 бит защиты, что остаётся надёжным.

AES-256 часто выбирается для приложений высокой безопасности и для обеспечения запаса прочности против будущих криптоаналитических достижений. Он также обеспечивает 128 бит защиты против квантовых компьютеров (против 64 бит у AES-128), что делает его рекомендуемым выбором для долгосрочной безопасности.

AES-192 используется редко — если вам нужно более 128 бит, вы обычно переходите сразу к 256. Дополнительная сложность AES-192 даёт мало преимуществ по сравнению с AES-128 для большинства моделей угроз.

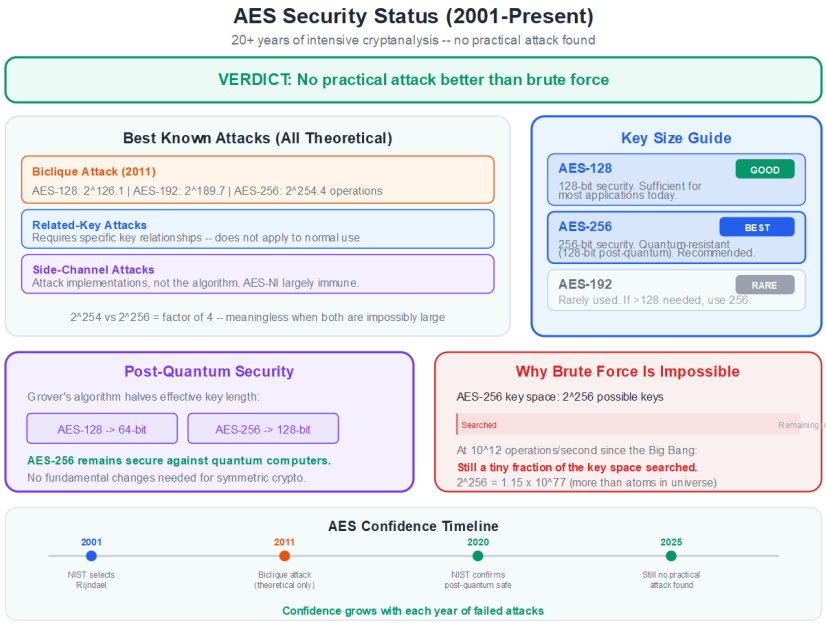


Рисунок 8: Состояние безопасности AES: размеры ключей и устойчивость к известным атакам

Режимы работы

Именно здесь многие реализации допускают ошибки. AES сам по себе шифрует только 16-байтовые блоки. Для шифрования более длинных сообщений необходимы режимы работы. Режим определяет, как объединяются блоки и как вносится рандомизация.

Позвольте подробно объяснить наиболее важные режимы.

ECB (электронная кодовая книга)

ECB — простейший режим: каждый блок шифруется независимо одним и тем же ключом.

Это почти всегда неправильно! Проблема в том, что идентичные блоки открытого текста порождают идентичные блоки шифротекста.

Классическая демонстрация — шифрование растрового изображения. Хотя значения отдельных пикселей зашифрованы, общий рисунок изображения остаётся различимым, поскольку похожие области (небо, трава, кожа) порождают похожий шифротекст. Знаменитое изображение "пингвин ECB" драматически показывает это — вы можете отчётливо видеть силуэт пингвина на зашифрованном изображении.

Рисунок 10: Пингвин Linux Tux, оригинальное изображение

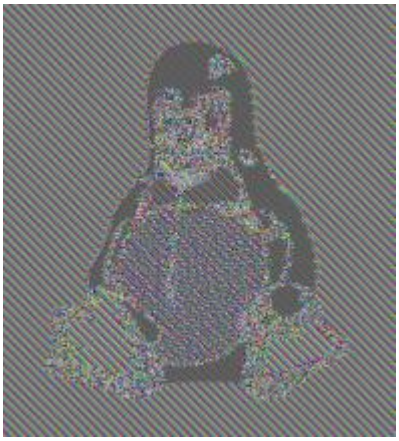


Рисунок 11: Пингвин Linux Tux, зашифрованный в режиме ECB, выявляющий паттерн

Эта слабость имеет реальные последствия для безопасности:

- Закономерности в данных раскрывают информацию
- Идентичные сообщения обнаруживаемы
- Возможны атаки повтором на уровне блоков

Никогда не используйте ECB для реального шифрования. Он существует в криптографических библиотеках, потому что является строительным блоком для других режимов и имеет нишевые применения (например, шифрование ровно одного блока), но никогда не должен использоваться для общего шифрования.

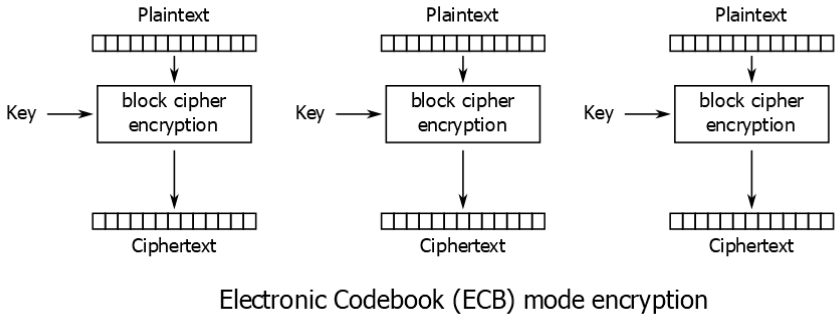


Рисунок 9: Режим ECB: независимое шифрование блоков (небезопасен для структурированных данных)

CBC (сцепление блоков шифротекста)

CBC устраняет слабость ECB путём сцепления блоков. Перед шифрованием каждого блока он XOR-ится с предыдущим блоком шифротекста. Первый блок XOR-ится со случайным вектором инициализации (IV).

Шифрование:

- C_0 = IV
- C_i = E(K, P_i XOR C_{i-1})

Расшифрование:

- P_i = D(K, C_i) XOR C_{i-1}

Это гарантирует, что идентичные блоки открытого текста шифруются в разные блоки шифротекста (поскольку они XOR-ятся с разными предыдущими блоками шифротекста). IV должен быть непредсказуемым для каждого сообщения — обычно случайным, хотя некоторые протоколы используют счётчики.

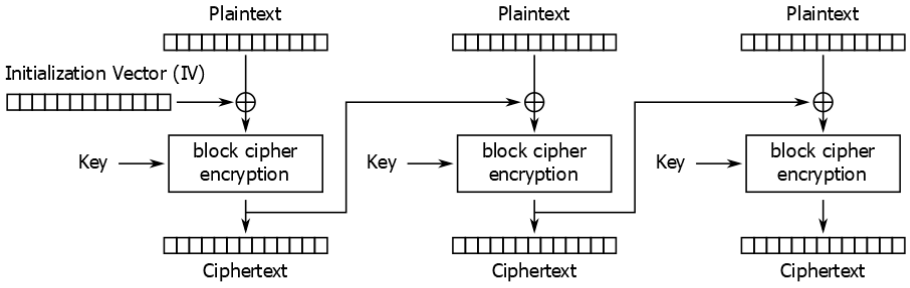
CBC обеспечивает хорошую безопасность, но имеет ряд недостатков:

Последовательное шифрование: каждый блок зависит от предыдущего, поэтому шифрование не может быть распараллелено. Расшифрование может быть распараллелено, поскольку требуется только текущий и предыдущий блоки шифротекста.

Распространение ошибок: повреждённый блок шифротекста затрагивает два блока открытого текста — повреждённый блок и следующий за ним.

Атаки на оракул дополнения: если злоумышленник может определить, имеет ли полученный шифротекст корректное дополнение, он зачастую может расшифровать всё сообщение. Это было убедительно продемонстрировано атакой POODLE на SSL 3.0 и атакой Lucky 13 на TLS.

IV должен быть случайным и непредсказуемым. Использование предсказуемого IV (например, счётчика) может позволить атаки с выбранным открытым текстом, при которых злоумышленники могут проверять свои предположения об открытом тексте.



Cipher Block Chaining (CBC) mode encryption

Рисунок 12: Режим CBC: сцепление блоков шифротекста с вектором инициализации

CTR (режим счётчика)

Режим CTR превращает блочный шифр в потоковый. Вместо прямого шифрования открытого текста шифруется значение счётчика (начиная с одноразового номера), и результат XOR-ится с открытым текстом.

Генерация гаммы:

- keystream_i = E(K, nonce || counter_i)

Шифрование/Расшифрование:

- C_i = P_i XOR keystream_i

Счётчик обычно начинается с 0 и увеличивается для каждого блока. В сочетании с уникальным одноразовым номером для каждого сообщения это обеспечивает уникальность гаммы.

Режим CTR имеет существенные преимущества:

Полное распараллеливание: и шифрование, и расшифрование могут быть полностью распараллелены, поскольку каждый блок обрабатывается независимо (после получения гаммы). Это даёт значительно более высокую пропускную способность на многоядерных системах.

Произвольный доступ: вы можете расшифровать любой блок независимо, вычислив его значение счётчика. Это полезно для шифрования дисков, где вы можете читать блок 1000, не читая блоки 1-999.

Не нужно дополнение: вы просто генерируете столько гаммы, сколько нужно, и выполняете XOR. Если ваше сообщение составляет 100 байт, вы генерируете 112 байт гаммы (7 блоков) и используете 100 байт.

Распространение ошибок на один бит: повреждённый бит шифротекста затрагивает только один бит открытого текста. Распространения ошибок не происходит.

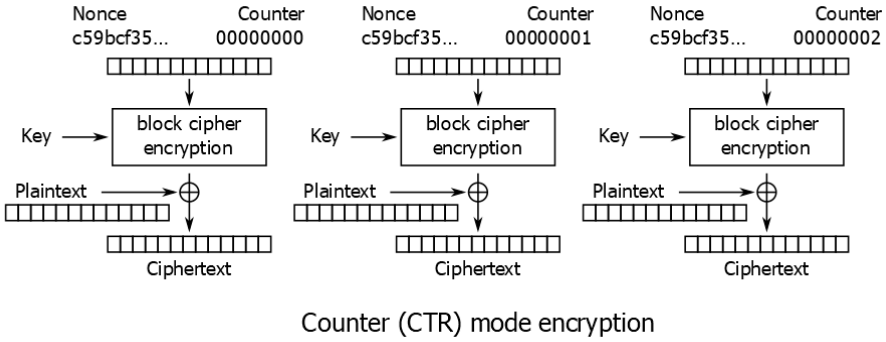


Рисунок 13: Режим CTR: параллельное шифрование на основе счётчика

Критическое требование — никогда не использовать одноразовый номер повторно с тем же ключом. Если вы зашифруете два сообщения с одним и тем же одноразовым номером:

- C_1 = P_1 XOR keystream
- C_2 = P_2 XOR keystream
- C_1 XOR C_2 = P_1 XOR P_2

Злоумышленник может выполнить XOR шифротекстов для получения XOR открытых текстов, что часто раскрывает значительную информацию.

GCM (режим Галуа со счётчиком)

GCM сочетает шифрование в режиме CTR с аутентификацией. Он гарантирует одновременно конфиденциальность и целостность, то есть злоумышленник не может модифицировать шифротекст без обнаружения.

Это то, что мы называем аутентифицированным шифрованием с присоединёнными данными (AEAD). Вы можете включить дополнительные данные (например, заголовки), которые аутентифицируются, но не шифруются.

GCM использует режим CTR для шифрования и полиномиальный хеш GHASH для аутентификации. Результатом является шифротекст плюс тег аутентификации (обычно 128 бит).

GCM — наиболее широко используемый режим аутентифицированного шифрования. Он применяется в:

- TLS 1.3 (обязательный набор шифров)
- TLS 1.2 (предпочтительный набор шифров)
- SSH
- IPSec
- Многочисленных прикладных протоколах

Требования и особенности GCM:

Уникальность одноразовых номеров: одноразовые номера никогда не должны повторяться с тем же ключом. Однократное повторное использование одноразового номера может полностью нарушить аутентификацию: злоумышленник может подделывать теги аутентификации и потенциально восстановить ключ аутентификации. Базовый принцип тот же, что вскрыл советский одноразовый блокнот в проекте ВЕНОНА: повторное использование криптографического материала (будь то ключ одноразового блокнота или одноразовый номер GCM) создаёт алгебраические зависимости, которые злоумышленник может использовать.

Размер одноразового номера: стандарт GCM использует 96-битные одноразовые номера. При случайных одноразовых номерах коллизии дней рождения становятся вероятными после приблизительно 2^48 сообщений. Для приложений с высокой нагрузкой используйте детерминированные одноразовые номера (счётчики) или алгоритмы с большими одноразовыми номерами.

Размер тега: рекомендуются 128-битные теги. Более короткие теги (96, 64, 32 бита) допускаются, но пропорционально снижают безопасность. С 64-битным тегом злоумышленник имеет шанс 2^-64 случайно подделать корректный тег.

Ограничения на размер сообщения: GCM рассчитан на сообщения до 2^36 блоков (около 64 ГБ). Более длинные сообщения следует разбивать на отдельно аутентифицируемые фрагменты.

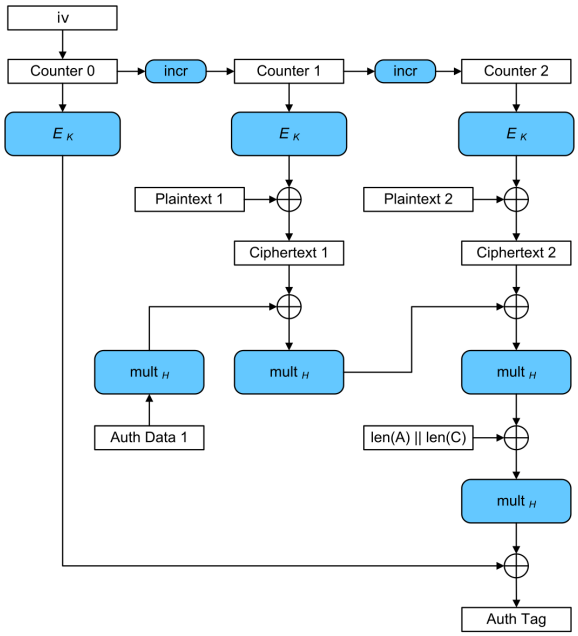


Рисунок 14: Режим GCM: Galois/Counter Mode, обеспечивающий шифрование и аутентификацию

Производительность: GCM очень быстр с аппаратным ускорением (инструкции CLMUL). Без аппаратной поддержки полиномиальное умножение GHASH относительно медленно.

CCM (счётчик с CBC-MAC)

CCM — ещё один режим аутентифицированного шифрования, сочетающий шифрование в режиме CTR с аутентификацией CBC-MAC. Он был разработан для ограниченных сред, таких как устройства IoT, и используется в:

- Bluetooth Low Energy
- ZigBee
- IEEE 802.15.4 (беспроводные сенсорные сети)
- WPA2 (WiFi)

CCM имеет иные компромиссы по сравнению с GCM:

Не требуется специальное оборудование: CBC-MAC использует те же операции AES, что и шифрование, тогда как GHASH в GCM требует другого оборудования (CLMUL) для оптимальной производительности.

Длина сообщения должна быть известна заранее: в отличие от GCM, который может обрабатывать данные в потоковом режиме, CCM необходимо знать общую длину сообщения до начала шифрования. Это связано с тем, что CBC-MAC вычисляется перед шифрованием.

Обычно медленнее GCM: на оборудовании с AES-NI и CLMUL GCM быстрее. На ограниченных устройствах без специализированного оборудования они могут быть сопоставимы.

Двухпроходная обработка: CCM требует двух проходов по данным — один для MAC, один для шифрования. GCM обрабатывает данные за один проход.

Сводка по выбору режима

Режим	Аутентификация	Распараллеливание	Дополнение	Основное применение
ECB	Нет	Да	Да	НИКОГДА НЕ ИСПОЛЬЗОВАТЬ
CBC	Нет	Только расшифрование	Да	Устаревшие системы, специальные задачи
CTR	Нет	Да	Нет	Строительный блок
GCM	Да	Да	Нет	AEAD общего назначения
CCM	Да	Нет	Нет	Ограниченные устройства

Рекомендация: для новых приложений всегда используйте аутентифицированное шифрование. GCM — выбор по умолчанию при наличии аппаратного ускорения. ChaCha20-Poly1305 (обсуждается далее) — отличный вариант при его отсутствии.

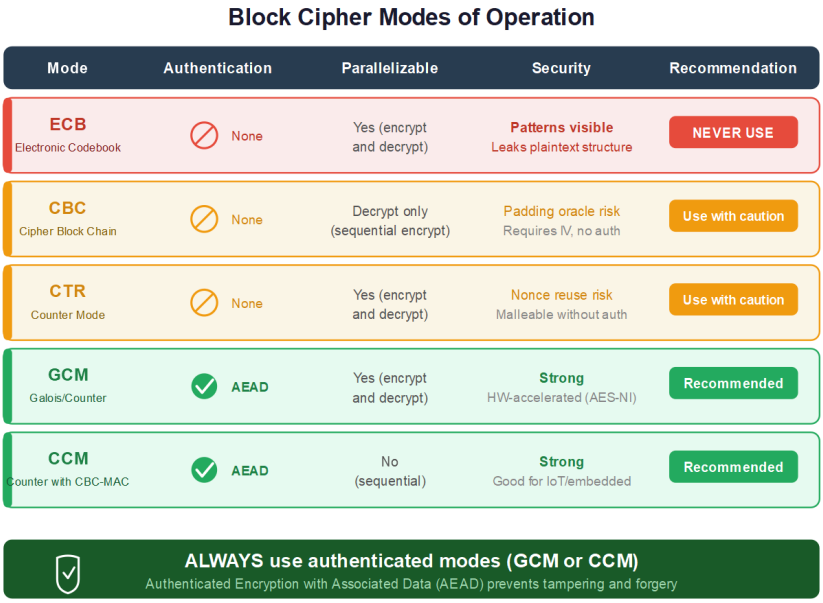


Рисунок 15: Сравнение режимов работы блочных шифров

Часть 5: ChaCha20 и ChaCha20-Poly1305

Происхождение ChaCha20

Хотя AES доминирует в аппаратных реализациях, он может быть медленным на устройствах без инструкций AES-NI. Операции поиска по S-блокам в программных реализациях также уязвимы к атакам по времени доступа к кэшу, что требует тщательных реализаций с постоянным временем выполнения, которые дополнительно снижают производительность.

Дэниел Бернштейн, плодовитый криптограф, известный практичными и элегантными решениями, разработал потоковый шифр Salsa20 в 2005 году для проекта eSTREAM. В 2008 году он опубликовал ChaCha20 — улучшенный вариант с лучшим рассеиванием при сохранении эффективности Salsa20.

ChaCha20 — потоковый шифр, генерирующий 512-битный блок гаммы из:

- 256-битного ключа
- 96-битного одноразового номера
- 32-битного счётчика блоков

Он использует только операции сложения, циклического сдвига и XOR (ARX), которые:

- Выполняются за постоянное время практически на всех процессорах (без обращений к таблицам)
- Просты в корректной реализации
- Достигают сильного рассеивания с меньшим числом раундов, чем шифры на основе S-блоков

Это делает ChaCha20 по своей природе устойчивым к атакам по побочным каналам на основе временных характеристик, даже без специальных мер в реализации.

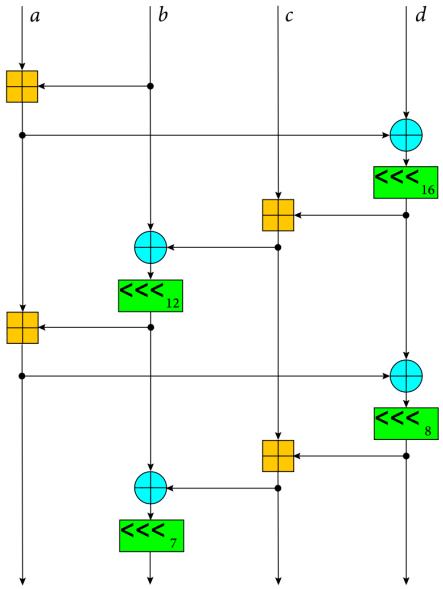


Рисунок 16: Потоковый шифр ChaCha20

Структура ChaCha20

Алгоритм поддерживает матрицу 4x4 из 32-битных слов (512 бит = 64 байта):

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

Где:

- c: четыре константных слова ("expand 32-byte k" в ASCII — константа типа "ничего в рукаве")
- k: восемь ключевых слов (256 бит)
- b: одно слово счётчика (32 бита)

- n: три слова одноразового номера (96 бит)

Функция четверть-раунда оперирует четырьмя словами:

```
a += b; d ^= a; d <<<= 16
c += d; b ^= c; b <<<= 12
a += b; d ^= a; d <<<= 8
c += d; b ^= c; b <<<= 7
```

Каждый раунд применяет эту четверть-раундовую функцию к:

- Сначала столбцам (столбцовые раунды)
- Затем диагоналям (диагональные раунды)

После 20 раундов (10 двойных раундов) исходное состояние прибавляется к перемешанному состоянию. Это сложение обеспечивает критическую нелинейность и гарантирует, что гамма не может быть инвертирована.

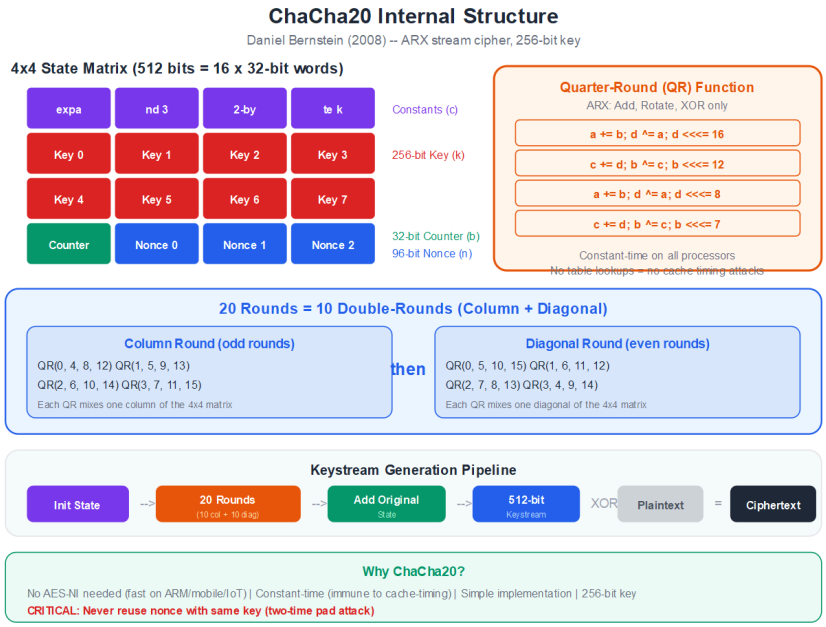


Рисунок 17: Структура четверть-раунда ChaCha20

Результирующий 512-битный блок является гаммой для данного значения счётчика. Для следующего блока счётчик увеличивается.

ChaCha20-Poly1305

Подобно тому как GCM сочетает AES-CTR с аутентификацией GHASH, ChaCha20-Poly1305 сочетает шифрование ChaCha20 с аутентификацией Poly1305. Эта комбинация стандартизирована в RFC 7539 (ныне RFC 8439) и в настоящее время широко развёрнута.

Poly1305 — одноразовый аутентификатор, разработанный Бернштейном. Он принимает 256-битный ключ и сообщение и вырабатывает 128-битный тег. Ключ

одноразовый — для каждого сообщения из гаммы ChaCha20 выводится новый ключ.

Как работает ChaCha20-Poly1305:

1. Генерируется первый блок гаммы (счётчик = 0)
2. Первые 256 бит используются как ключ Poly1305
3. Открытый текст шифруется ChaCha20 (счётчик начинается с 1)
4. Poly1305 вычисляется по присоединённым данным и шифротексту
5. На выходе — шифротекст и 128-битный тег

Poly1305 чрезвычайно быстр — он обрабатывает данные большими фрагментами, используя только умножение и сложение по модулю простого числа. Требование одноразового ключа удовлетворяется выводом нового ключа для каждого сообщения.

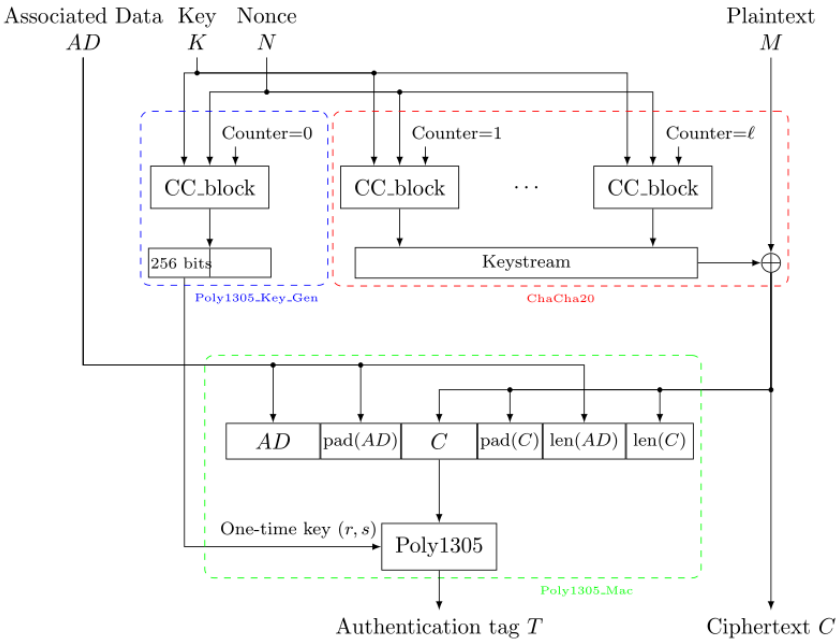


Рисунок 18: ChaCha20-Poly1305: аутентифицированное шифрование

Преимущества ChaCha20-Poly1305:

- **Быстр без аппаратного ускорения**: часто в 3-5 раз быстрее AES-GCM в чистой программной реализации на мобильных устройствах
- **Постоянное время по конструкции**: отсутствие обращений к таблицам исключает атаки по времени доступа к кэшу
- **Простая реализация**: труднее реализовать неправильно
- **Стандартизирован**: RFC 8439, обязательный набор шифров TLS 1.3, широко поддерживается

ChaCha20-Poly1305 используется в:

- TLS 1.3 (обязательная реализация)

- WireGuard VPN
- Протоколе Signal
- Google Chrome и Android
- Многих мобильных приложениях

XChaCha20

Стандартный ChaCha20 использует 96-битный одноразовый номер. Хотя этого достаточно для большинства применений, он требует тщательного управления одноразовыми номерами — случайные одноразовые номера имеют риск коллизии после 2^48 сообщений из-за границы дней рождения.

XChaCha20 расширяет одноразовый номер до 192 бит, делая генерацию случайных одноразовых номеров безопасной для практически неограниченного числа сообщений. Он работает следующим образом:

1. Первые 128 бит одноразового номера используются с HChaCha20 (вариант ChaCha20, выводящий основное состояние) для получения 256-битного подключа
2. Оставшиеся 64 бита используются как одноразовый номер для стандартного ChaCha20 с подключом

Эта конструкция доказуемо безопасна, если безопасен ChaCha20.

XChaCha20-Poly1305 рекомендуется для:

- Распределённых систем, где координация одноразовых номеров затруднена
- Высоконагруженного шифрования, где предел в 2^48 сообщений слишком мал
- Приложений, использующих случайные одноразовые номера
- Шифрования файлов, где обеспечение уникальных одноразовых номеров для каждого файла затруднительно

Библиотека libsodium реализует XChaCha20-Poly1305 как рекомендуемую конструкцию AEAD.

AES-GCM и ChaCha20-Poly1305 в сравнении

Оба являются превосходными алгоритмами аутентифицированного шифрования. Выбор зависит от вашей среды:

Предпочитайте AES-GCM, когда:

- Доступно аппаратное ускорение (AES-NI, CLMUL)
- Критична максимальная пропускная способность
- Требования соответствия предписывают алгоритмы NIST

- Необходима совместимость с устаревшими системами

Предпочитайте ChaCha20-Poly1305, когда:

- Работаете на устройствах без AES-NI (мобильные, встраиваемые, старые системы)
- Критична устойчивость к атакам по побочным каналам
- Ценится простота реализации
- Используются случайные одноразовые номера (XChaCha20)

На практике многие системы поддерживают оба и согласовывают наилучший вариант. TLS 1.3 требует реализации обоих.

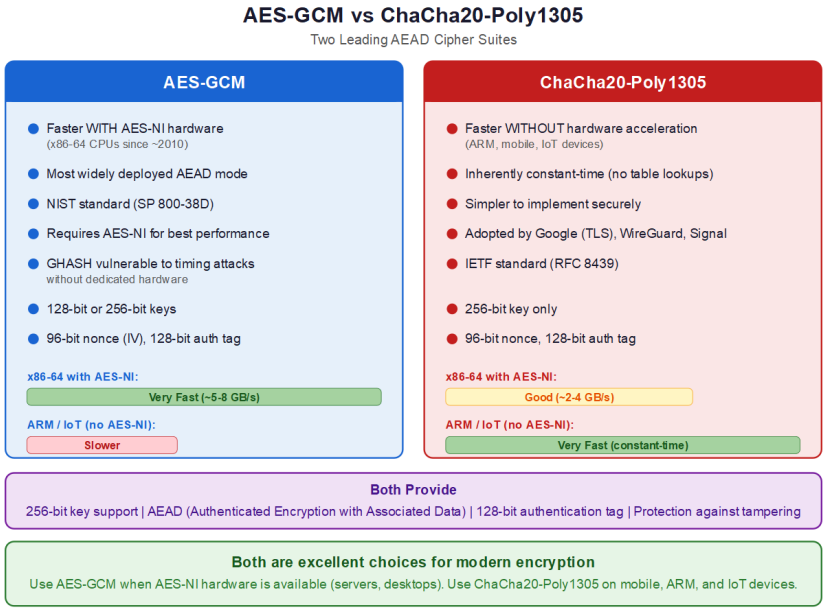


Рисунок 19: AES-GCM и ChaCha20-Poly1305: сравнение производительности

Часть 6: Ascon — облегчённая криптография

Потребность в облегчённой криптографии

Интернет вещей породил спрос на криптографию, работающую на крайне ограниченных устройствах:

- Датчики с килобайтами оперативной памяти
- Микроконтроллеры, питающиеся от монетных батарей
- RFID-метки с минимальной схемотехникой
- Промышленные системы управления с требованиями реального времени

Стандартные алгоритмы, такие как AES, хотя и эффективны на современных процессорах, могут быть сложными для этих устройств:

- AES требует значительного объёма кода и оперативной памяти для таблиц подстановки
- Реализации AES с постоянным временем ещё крупнее
- Умножение GHASH в GCM дорогостояще без аппаратной поддержки

В 2019 году NIST запустил конкурс облегчённой криптографии для стандартизации алгоритмов, оптимизированных для ограниченных сред. После многолетней оценки, начавшейся с 57 первоначальных заявок, NIST объявил Ascon победителем в феврале 2023 года.

Конструкция Ascon

Ascon был разработан Кристофом Добрауником, Марией Айхлзедер, Флорианом Менделем и Мартином Шлаффером из Грацского технического университета и компании Infineon Technologies, производителя полупроводников. Конструкция отдаёт приоритет:

- Малому размеру состояния (320 бит)
- Простым операциям (XOR, AND, циклический сдвиг)
- Гибкости для 8-битных, 32-битных и 64-битных платформ
- Устойчивости к атакам по побочным каналам

Ascon использует конструкцию губки, аналогичную SHA-3, но оптимизированную для малых реализаций. Ядро — 320-битная перестановка, оперирующая пятью 64-битными словами.

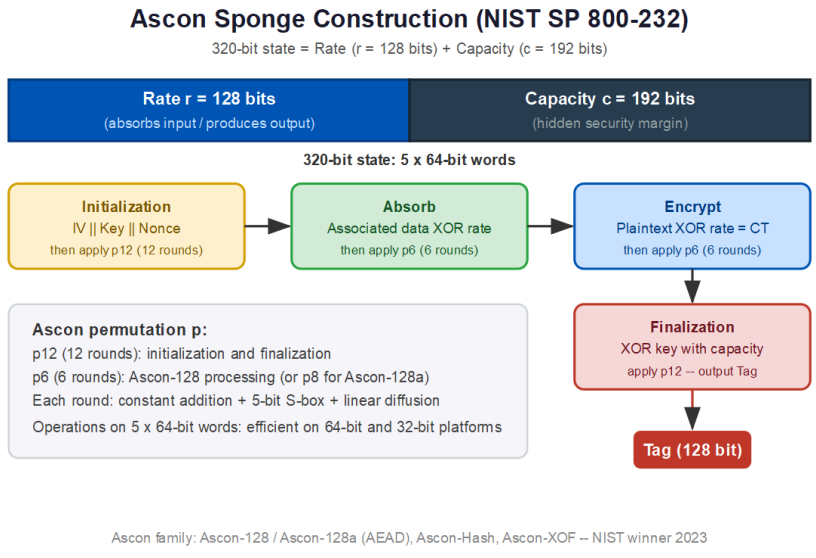


Рисунок 20: Конструкция губки Ascon: 320-битное состояние со скоростью 128 бит и ёмкостью 192 бита

Перестановка использует простую раундовую функцию с:

- Добавлением раундовых констант (предотвращение симметрии)
- Слоем подстановки (5-битный S-блок, применяемый к 64 параллельным срезам)
- Линейным слоем рассеивания (побитовый циклический сдвиг и XOR)

Несмотря на то что конструкция ориентирована на 64-битные слова, операции могут быть эффективно реализованы в битовом формате для 8-битных и 32-битных платформ. Это означает, что один и тот же алгоритм хорошо работает на совершенно различном оборудовании.

Варианты Ascon

Ascon предоставляет семейство алгоритмов:

Аутентифицированное шифрование:

- **Ascon-128**: 128-битный ключ, 128-битный одноразовый номер, 128-битный тег, стандартный уровень безопасности
- **Ascon-128a**: та же безопасность, но быстрее (меньше раундов), более высокая пропускная способность

Хеш-функции:

- **Ascon-Hash**: 256-битная хеш-функция
- **Ascon-Hasha**: более быстрый вариант с меньшим числом раундов
- **Ascon-XOF**: функция с расширяемым выходом (произвольная длина вывода)

MAC:

- **Ascon-Mac**: код аутентификации сообщений
- **Ascon-PRF**: псевдослучайная функция

Наличие AEAD, хеширования и MAC из одного семейства алгоритмов упрощает реализацию и уменьшает объём кода — что важно на ограниченных устройствах.

Производительность Ascon

Ascon демонстрирует впечатляющую производительность на ограниченных устройствах:

На 8-битных микроконтроллерах AVR:

- Шифрование: приблизительно 15 тактов/байт
- Хеширование: приблизительно 20 тактов/байт

На 32-битных ARM Cortex-M:

- Шифрование: приблизительно 10 тактов/байт

- Конкурентоспособен с AES даже без аппаратной поддержки AES

На 64-битных платформах:

- Очень быстр, хотя AES-GCM с аппаратной поддержкой быстрее

Малый размер состояния (320 бит = 40 байт) означает, что Ascon требует минимальной оперативной памяти, что важно для устройств с килобайтами памяти.

Когда использовать Ascon

Ascon предназначен для ограниченных сред, а не для замены AES-GCM на серверах. Используйте Ascon, когда:

- Работаете с устройствами IoT, встраиваемыми системами или смарт-картами
- Аппаратные ресурсы (вентили, память, объём кода) сильно ограничены
- Важна устойчивость к атакам по побочным каналам, а аппаратные контрмеры недоступны
- Вам нужно одно семейство алгоритмов для шифрования, хеширования и MAC
- Критично энергопотребление (устройства с батарейным питанием)

Для стандартных серверов, настольных компьютеров и мобильных устройств с современным оборудованием AES-GCM и ChaCha20-Poly1305 остаются рекомендуемым выбором благодаря аппаратному ускорению и зрелым реализациям.

Ascon стандартизирован в специальной публикации NIST SP 800-232 (ожидается в 2024 году) и будет обязательным для приложений IoT правительства США.

Часть 7: Управление ключами

Проблема управления ключами

Мы обсудили множество превосходных алгоритмов шифрования, но я должен подчеркнуть нечто критически важное: большинство криптографических отказов вызвано не взломом алгоритмов. Они вызваны неправильным управлением ключами. Мы уже видели эту закономерность дважды в данной лекции: Энигма была вскрыта не из-за слабости её подстановочного механизма, а из-за эксплуатационных провалов в процедурах обращения с ключами; советский одноразовый блокнот был вскрыт в проекте ВЕНОНА не из-за какого-либо дефекта шифра, а из-за того, что фабрика дублировала ключевой материал под давлением военного времени.

Вы можете использовать AES-256-GCM — золотой стандарт — и всё равно иметь совершенно незащищённое шифрование, если:

- Ключ получен из слабого пароля
- Ключ хранится в открытом виде в файле конфигурации
- Один и тот же ключ используется годами без ротации
- Ключи передаются по незащищённому каналу
- Генерация ключей использует слабый генератор случайных чисел

Согласно анализу Schneier on Security и исследованиям IEEE, проблемы управления ключами являются причиной значительно большего числа реальных нарушений безопасности, чем алгоритмические слабости. Алгоритм обычно является самым надёжным звеном; всё остальное вокруг него слабее.

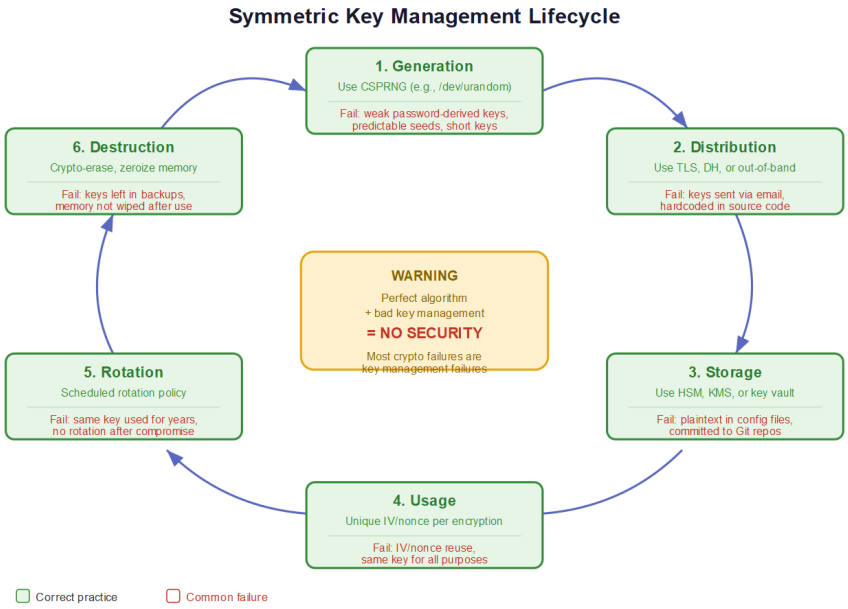


Рисунок 21: Управление ключами: генерация, распределение, хранение и ротация

Позвольте обсудить инструменты и практики надлежащего управления ключами.

Генерация ключей

Ключи должны генерироваться с помощью криптографически стойких генераторов случайных чисел (CSPRNG). На современных операционных системах:

- Linux/macOS: /dev/urandom, системный вызов getrandom()
- Windows: CryptGenRandom(), BCryptGenRandom()

Никогда не используйте:

- Стандартные библиотечные функции для случайных чисел (rand(), random())
- Текущее время в качестве начального значения

- Предсказуемые источники, такие как идентификаторы процессов
- Источники с недостаточной энтропией

Код генерации ключей должен быть прост: вызовите CSPRNG операционной системы и запросите необходимое количество байт. Библиотеки вроде libsodium предоставляют функции randombytes(), которые делают это правильно.

Для AES-256 вам нужны 32 криптографически случайных байта. Вот и всё. Не усложняйте.

Вывод ключей: HKDF

Функция вывода ключей на основе HMAC (HKDF), стандартизированная в RFC 5869, используется для получения криптографических ключей из другого ключевого материала. Она особенно полезна для:

- Вывода нескольких ключей из общего секрета (например, после обмена Диффи-Хеллмана)
- Расширения коротких ключей в более длинные
- Добавления контекста к выводу ключей

HKDF работает в два этапа:

1. Извлечение (Extract): принимает исходный ключевой материал (IKM) и необязательную соль, вырабатывая псевдослучайный ключ (PRK). Это концентрирует энтропию из возможно неравномерного входа.

```
PRK = HMAC(salt, IKM)
```

2. Расширение (Expand): принимает PRK и необязательную контекстную информацию, вырабатывая выходной ключевой материал (OKM) любой желаемой длины.

```
OKM = HMAC(PRK, info || 0x01) || HMAC(PRK, T1 || info || 0x02) || ...
```

Соль должна быть случайной, если это возможно, но даже несекретная постоянная соль улучшает безопасность по сравнению с её отсутствием. Параметр info позволяет выводить разные ключи для разных целей из одного и того же входа — это называется разделением доменов.

Пример использования: после обмена ключами Диффи-Хеллмана, порождающего общий секрет, используйте HKDF для вывода отдельных ключей для шифрования и аутентификации:

```
encryption_key = HKDF(shared_secret, salt, "encryption")
authentication_key = HKDF(shared_secret, salt, "authentication")
```

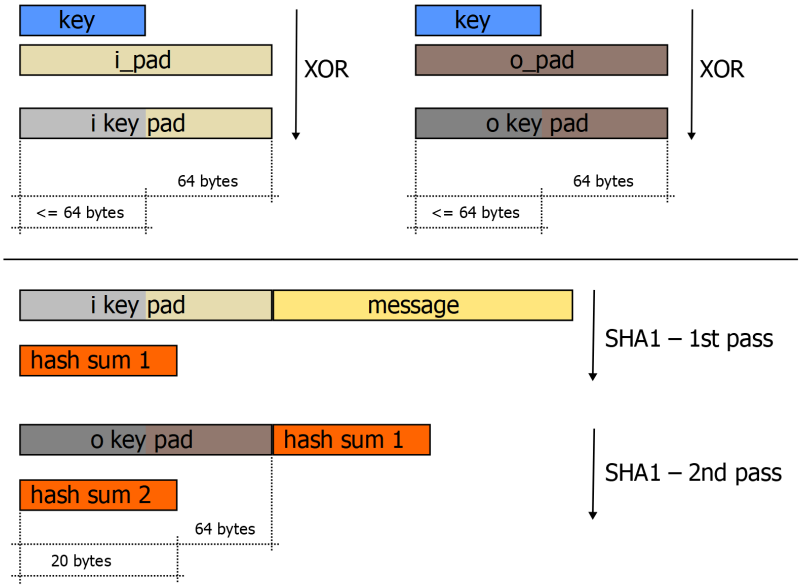


Рисунок 22: HMAC: код аутентификации сообщения на основе хеш-функции

Вывод ключей из паролей: PBKDF2

Пароли непригодны для прямого использования в качестве криптографических ключей:

- Они имеют недостаточную энтропию (обычно 20-40 бит против необходимых 128-256 бит)
- Они уязвимы к словарным атакам
- Распространённые пароли встречаются часто

Password-Based Key Derivation Function 2 (PBKDF2), стандартизированная в RFC 8018, преобразует пароли в ключи, при этом намеренно замедляя вычисления.

PBKDF2 применяет псевдослучайную функцию (обычно HMAC-SHA256) к паролю множество раз — десятки или сотни тысяч итераций. Это количество итераций (фактор стоимости) делает атаки перебором дорогостоящими.

```
DK = PBKDF2(password, salt, iterations, output_length)
```

Соль (не менее 128 бит, случайная для каждого пароля) предотвращает атаки с помощью радужных таблиц и гарантирует, что одинаковые пароли порождают разные ключи.

Однако PBKDF2 имеет ограничения:

- Не требует значительного объёма памяти, поэтому GPU могут эффективно атаковать её
- Легко распараллеливается

- Каждая итерация HMAC относительно дешева на специализированном оборудовании

Для хеширования паролей в частности современные алгоритмы вроде Argon2id превосходят PBKDF2, так как требуют значительного объёма памяти и устойчивы к ускорению на GPU/ASIC. Хеширование паролей мы подробно рассмотрим в Лекции 11.

PBKDF2 остаётся уместной, когда:

- Необходима совместимость со старыми системами
- Спецификация протокола требует её (например, WPA2)
- Функции с требованием памяти недоступны

При использовании PBKDF2 применяйте не менее 310 000 итераций для HMAC-SHA256 (рекомендация OWASP (Open Web Application Security Project, проект по безопасности веб-приложений) 2023).

bcrypt

В то время как PBKDF2 — универсальная функция вывода ключей, **bcrypt** был разработан специально для хеширования паролей. Он включает встроенную 128-битную соль и настраиваемый фактор стоимости, управляющий временем вычисления. bcrypt, scrypt и Argon2id мы подробно рассмотрим в Лекции 11.

Соль и перец

Термин "соль" в криптографии восходит к статье Роберта Морриса и Кена Томпсона "Password Security: A Case History" (1979), где они ввели случайные данные, подмешиваемые в хеши паролей Unix для защиты от предвычисленных атак. Название основано на общей английской метафоре: подобно тому как добавление небольшого количества соли меняет характер блюда, добавление случайных данных к паролю перед хешированием меняет результирующий хеш. Термин "перец" появился позднее по аналогии, распространяя кулинарную метафору на секретное значение, хранящееся отдельно от хеша. Сама концепция старше названия: Стивен Белловин предложил секретный "локальный параметр" в посте на Bugtraq в 1995 году, а Уди Манбер описал аналогичную схему в 1996 году, назвав её "секретная соль." Название "перец" закрепилось как естественный кулинарный спутник соли, подчёркивая, что соль хранится рядом с хешем (публична), а перец хранится отдельно (секретен).

Оба приёма связаны с более широким понятием **хеширования с ключом**: обычная хеш-функция вычисляет H(данные), и любой с теми же данными может воспроизвести результат; хеширование с ключом включает секретное значение, поэтому только тот, кто знает секрет, может получить или проверить результат. Соль не является ключом (она публична), но перец выполняет роль секретного

входного значения, делая хеширование пароля с перцем формой хеширования с ключом. Хеширование с ключом (HMAC и коды аутентификации сообщений) мы подробно рассматриваем в Лекции 11.

Все функции хеширования паролей используют **соль**: случайное значение (не менее 128 бит), уникальное для каждого пароля и хранящееся рядом с хешем. Соль выполняет две задачи:

1. **Предотвращает атаки с помощью радужных таблиц**: предвычисленные таблицы соответствия распространённых паролей хешам бесполезны, поскольку каждый пароль имеет свою соль, порождая различный хеш даже для одинаковых паролей.
2. **Предотвращает многоцелевые атаки**: без соли злоумышленник, вычислив хеш одного предполагаемого пароля, может проверить его сразу по всем хешам в базе данных. С индивидуальной солью каждое предположение должно вычисляться отдельно для каждого пользователя.

Все современные хеш-функции для паролей (bcrypt, Argon2id) генерируют и встраивают соль автоматически. Вам никогда не нужно управлять солью вручную.

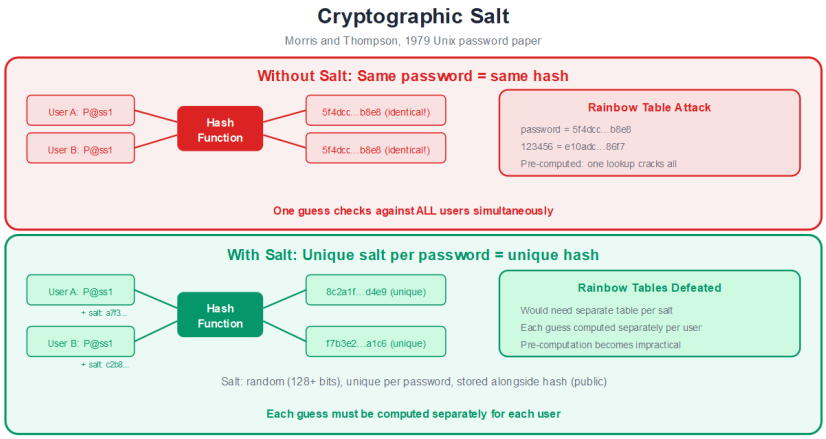


Рисунок 23: Криптографическая соль: предотвращение атак с радужными таблицами

Перец добавляет второй уровень защиты поверх соли. Перец (pepper) — это секрет уровня приложения, общий для всех паролей в приложении, но хранящийся отдельно от базы данных: в файле конфигурации приложения, переменной среды или аппаратном модуле безопасности (HSM). Перец никогда не хранится в базе данных.

Проверка пароля с перцем работает так: `hash = bcrypt(pepper + password, salt)`. Перец конкатенируется с паролем перед хешированием.

Почему перец + соль лучше, чем одна соль: если злоумышленник похитит базу данных (через SQL-инъекцию, кражу резервной копии или инсайдерский доступ), он получит все соли и хеши паролей. При наличии только соли он может немедленно начать офлайн-атаку перебором на каждый хеш. С перцем он этого не сможет: для каждого вычисления хеша необходим перец, а перца нет в базе данных. Злоумышленнику потребуется отдельно скомпрометировать сервер приложения или его конфигурацию, чтобы получить перец. Это эшелонированная защита: база данных не является единственной точкой отказа для безопасности паролей.

Практические соображения относительно перца:

- Если перец утерян (конфигурация удалена, отказ HSM), все пароли становятся неверифицируемыми и все пользователи должны сбросить свои пароли
- Ротация (смена) перца требует повторного хеширования всех паролей, что означает необходимость входа пользователей со старым перцем, чтобы система могла перехешировать с новым
- Некоторые считают, что если злоумышленник имеет доступ к базе данных, он, вероятно, имеет и доступ к конфигурации приложения; однако SQL-инъекция даёт прямой доступ к базе данных, не обязательно предоставляя доступ к файловой системе или переменным среды

Перец рекомендуется для приложений с высокими требованиями к безопасности, где кража базы данных является реальной моделью угрозы. Для большинства приложений индивидуальная соль с надёжным алгоритмом (Argon2id или bcrypt) и высоким фактором стоимости обеспечивает достаточную защиту.

Хранение ключей

Производственные системы никогда не должны хранить криптографические ключи в открытом виде. Варианты включают:

Аппаратные модули безопасности (HSM):

- Специализированные аппаратные устройства, хранящие ключи и выполняющие криптографические операции
- Ключи никогда не покидают HSM в открытом виде
- Защищённая от несанкционированного доступа физическая конструкция
- Используются финансовыми организациями, удостоверяющими центрами, облачными провайдерами
- Примеры: Thales Luna (Thales, поставщик решений кибербезопасности и защиты данных), AWS CloudHSM

Сервисы управления ключами (KMS):

- Облачные сервисы, обеспечивающие хранение ключей на основе HSM

- AWS KMS, Azure Key Vault, GCP Cloud KMS
- Журналирование аудита, управление доступом, автоматическая ротация
- Управление ключами через API
- Экономичная альтернатива выделенным HSM

Безопасные анклавы:

- Возможности процессора, создающие изолированные среды выполнения
- Intel SGX (Software Guard Extensions)
- ARM TrustZone
- Ключи защищены даже от привилегированного системного ПО
- Подвержены атакам по побочным каналам (Foreshadow и др.)

Хранилища ключей операционной системы:

- Windows DPAPI (Data Protection API)
- macOS Keychain
- Связка ключей ядра Linux
- Ключи шифруются учётными данными пользователя или системы
- Подходят для настольных приложений

Зашифрованная конфигурация:

- Ключи шифруются мастер-ключом
- Мастер-ключ защищён одним из вышеперечисленных способов
- Распространено в разработке и приложениях с более низкими требованиями к безопасности

Никогда не:

- Храните ключи в исходном коде
- Храните ключи в файлах конфигурации в открытом виде
- Храните ключи в переменных среды (утечка в журналах, сообщениях об ошибках)
- Фиксируйте ключи в системах контроля версий (используйте управление секретами)

Ротация ключей

Ключи должны периодически заменяться. Почему?

1. Ограничение ущерба: если ключ скомпрометирован, ротация ограничивает объём затронутых данных. Данные, зашифрованные до компрометации и после ротации ключей, остаются защищёнными.

2. Криптографическая гигиена: некоторые режимы имеют ограничения на объём данных, который следует шифровать одним ключом:

- GCM: приблизительно 2^32 блока (64 ГБ) на один одноразовый номер, 2^32 одноразовых номера на ключ
- После этих пределов гарантии безопасности ослабевают

3. Требования соответствия: многие стандарты требуют периодической ротации ключей:

- PCI-DSS требует ежегодной ротации ключей шифрования
- Некоторые нормативные акты устанавливают максимальный срок жизни ключей

4. Переход на новые алгоритмы: регулярная ротация упрощает переход на новые алгоритмы или ключи большего размера.

Хорошая система управления ключами поддерживает версионирование:

- Шифруйте новые данные последней версией ключа
- Расшифровывайте данные той версией ключа, которой они были зашифрованы
- Постепенно перешифровывайте старые данные новыми ключами
- В конечном итоге выводите из эксплуатации старые версии ключей

Это позволяет выполнять плавные переходы без сбоев.

Иерархии ключей

Крупные системы часто используют иерархии ключей:

Мастер-ключи (ключи шифрования ключей):

- Хранятся в HSM или KMS
- Используются для шифрования других ключей
- Заменяются нечасто, с тщательным планированием

Ключи шифрования данных (DEK):

- Шифруют реальные данные
- Шифруются мастер-ключами при хранении
- Заменяются часто
- Могут быть уникальными для каждой записи или сеанса

Сеансовые ключи:

- Используются для одного сеанса связи
- Выводятся из обмена ключами (Диффи-Хеллман)
- Уничтожаются после завершения сеанса

Эта иерархия обеспечивает эшелонированную защиту и гибкость. Если DEK скомпрометирован, затронуты только данные, зашифрованные этим ключом, а не вся система.

Часть 8: Лучшие практики и рекомендации

Позвольте завершить конкретными рекомендациями по использованию симметричного шифрования в 2025-2026 годах.

Выбор алгоритма

1. Для аутентифицированного шифрования общего назначения: AES-256-GCM или ChaCha20-Poly1305. Оба являются превосходным выбором.

- Предпочитайте AES-GCM при наличии аппаратного ускорения (AES-NI)
- Предпочитайте ChaCha20-Poly1305 на мобильных устройствах без AES-NI или когда критична реализация с постоянным временем
- Оба обязательны в TLS 1.3 — если вы поддерживаете TLS, оба уже доступны

2. Для ограниченных устройств IoT: Ascon-128 или Ascon-128a

- Стандартизированы NIST для этой цели
- Эффективны даже на 8-битных микроконтроллерах
- Одно семейство алгоритмов для шифрования и хеширования

3. Для шифрования множества сообщений со случайными одноразовыми номерами: XChaCha20-Poly1305

- 192-битный одноразовый номер устраняет проблемы дней рождения
- Безопасен для практически неограниченного числа сообщений
- Доступен в libsodium

4. Никогда не используйте:

- DES (56-битный ключ, тривиально взламывается)
- 3DES (медленный, устаревший, конец эксплуатации в 2030 году)
- RC4 (множество практических атак)
- Режим ECB (видны закономерности)
- Собственные или проприетарные алгоритмы

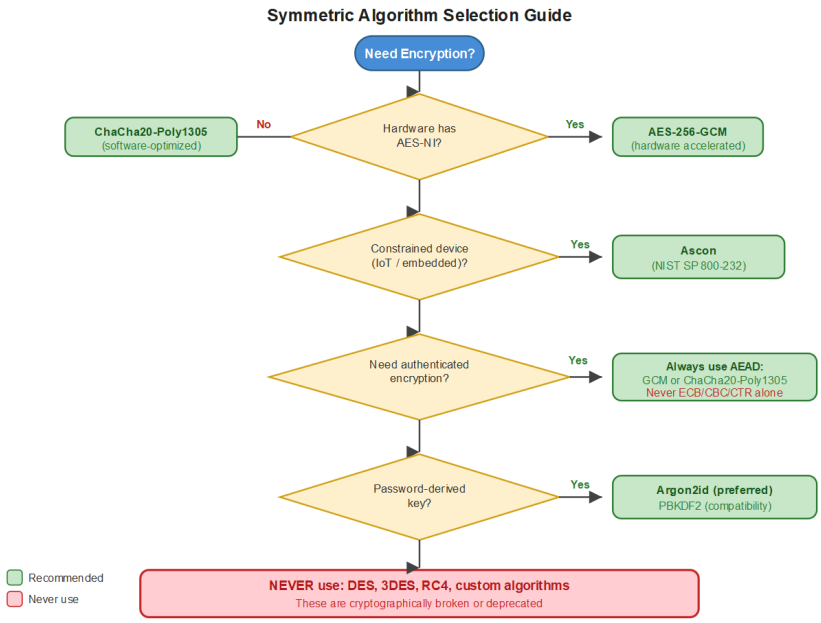


Рисунок 24: Руководство по выбору алгоритмов симметричной криптографии

Рекомендации по реализации

1. Используйте проверенные библиотеки: не реализуйте криптографические алгоритмы самостоятельно. Используйте:

- libsodium (превосходный дизайн API, устойчивость к неправильному использованию)
- OpenSSL (всеобъемлющий, широко развёрнут)
- BoringSSL (ответвление OpenSSL от Google, более чистый код)
- Платформенные библиотеки (.NET, Java Cryptography Architecture)
- Пакеты Go crypto/x

Даже эксперты допускают ошибки при реализации. Реализация с постоянным временем, устойчивость к побочным каналам и корректное дополнение — тонкие задачи. Доверяйте аудированному коду.

2. Всегда используйте аутентифицированное шифрование: простое шифрование (без аутентификации) позволяет злоумышленникам модифицировать шифротекст без обнаружения. Это сделало возможными разрушительные атаки:

- Атаки на оракул дополнения
- Атаки подменой битов
- Атаки на путаницу форматов

Всегда используйте GCM, CCM или ChaCha20-Poly1305. Никогда не используйте CBC, CTR или другие режимы без аутентификации для шифрования без добавления отдельной аутентификации.

3. Никогда не используйте одноразовые номера повторно: это наиболее распространённая ошибка реализации. Повторное использование одноразового номера в GCM полностью нарушает аутентификацию.

- Используйте счётчики для последовательных систем (обеспечьте сохранность между перезапусками)
- Используйте случайные одноразовые номера с XChaCha20 для распределённых систем
- При необходимости отслеживайте одноразовые номера для обнаружения повторного использования

4. Выводите ключи правильно:

- Используйте HKDF для вывода ключей из источников с высокой энтропией
- Используйте Argon2id для паролей (рассматривается в Лекции 11)
- Добавляйте разделение доменов (разные строки info для разных целей)

5. Планируйте криптографическую гибкость: проектируйте системы так, чтобы алгоритмы можно было менять без существенных модификаций:

- Включайте идентификаторы алгоритмов в форматы зашифрованных данных
- Используйте уровни абстракции в коде
- Не встраивайте выбор алгоритма жёстко по всей кодовой базе

Распространённые ошибки реализации

Позвольте выделить ошибки, которые я наблюдал в реальных системах:

Использование режима ECB: обычно потому, что он является режимом по умолчанию в некоторых библиотеках или потому что разработчики не понимают режимов.

Повторное использование одноразовых номеров: использование постоянного одноразового номера, повторное использование одноразовых номеров после перезапуска (отсутствие постоянного счётчика) или недостаточная длина случайного одноразового номера.

Жёстко закодированные ключи: ключи в исходном коде, зафиксированные в репозиториях, обнаруживаемые путём обратной разработки.

Слабый вывод ключей: использование хеша MD5 или SHA-256 от пароля без итераций. Использование временной метки в качестве ключа. Недостаточная соль.

Отсутствие аутентификации: использование AES-CBC или AES-CTR без HMAC, допускающее модификацию шифротекста.

Короткие теги: использование 32-битных или 64-битных тегов аутентификации для экономии места, радикально снижающее устойчивость к подделке.

Игнорирование проверки тега: проверка тега после расшифрования вместо до, или полное отсутствие проверки.

Генерация случайных чисел: использование предсказуемых источников, неправильная повторная инициализация, недостаточный сбор энтропии.

Взгляд в будущее: постквантовые соображения

Симметричное шифрование хорошо подготовлено к постквантовой эре. Алгоритм Гровера, работающий на достаточно мощном квантовом компьютере, может осуществить поиск среди 2^n возможностей за 2^(n/2) операций. Это фактически уменьшает вдвое безопасность симметричных алгоритмов.

Ответ прост: используйте 256-битные ключи.

- AES-256 предоставляет 128 бит постквантовой безопасности, что более чем достаточно
- ChaCha20 с его 256-битными ключами аналогично защищён
- Симметричное шифрование не требует принципиальных изменений для квантовой эры

Вот почему AES-256 и ChaCha20 (оба с 256-битными ключами) являются рекомендуемым выбором — они уже устойчивы к квантовым вычислениям с комфортным запасом безопасности.

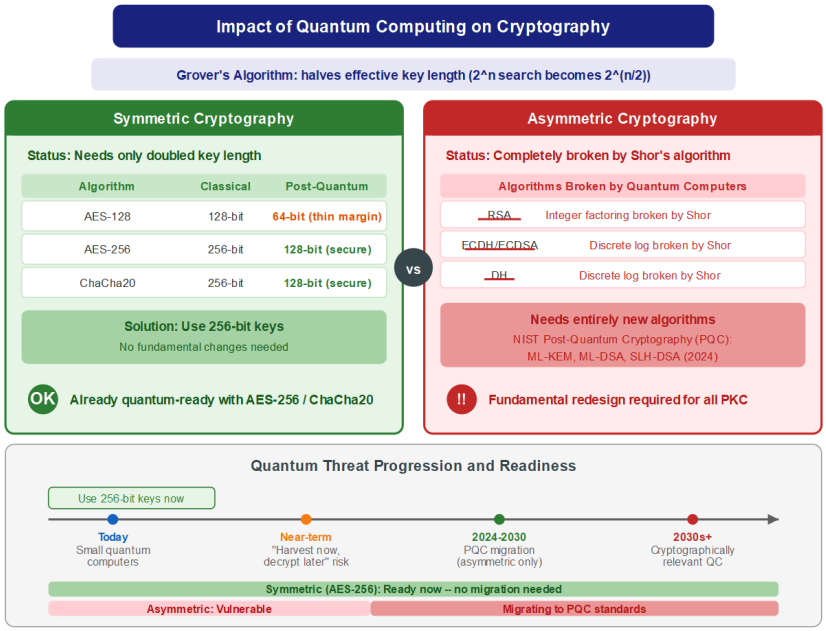


Рисунок 25: Постквантовая криптография: подготовка к угрозам квантовых вычислений

Асимметричное шифрование стоит перед значительно большей проблемой, как мы обсудим на следующей неделе. Но ваш сегодняшний выбор симметричного шифрования останется безопасным против квантовых компьютеров.

Заключение

Сегодня мы рассмотрели основы симметричного шифрования — от принципов Шеннона о перемешивании и рассеивании до современных алгоритмов, таких как AES, ChaCha20 и Ascon.

Ключевые выводы:

1. **Используйте аутентифицированное шифрование** (AES-GCM или ChaCha20-Poly1305) с уникальными одноразовыми номерами. Простое шифрование без аутентификации опасно.
2. **AES-256 обеспечивает превосходную безопасность**, в том числе против будущих квантовых компьютеров. Он является золотым стандартом не без причины.
3. **ChaCha20-Poly1305 — превосходная альтернатива**, особенно когда аппаратное ускорение недоступно или когда критична реализация с постоянным временем.
4. **Ascon — новый стандарт для ограниченных устройств**, обеспечивающий эффективное аутентифицированное шифрование на IoT и встраиваемых системах.
5. **DES и 3DES устарели**. Никогда не используйте их в новых системах и планируйте миграцию с устаревших систем.
6. **Правильный вывод и управление ключами критически важны**. Хорошие алгоритмы с плохим управлением ключами не обеспечивают безопасности, как демонстрируют случаи Энигмы и ВЕНОНЫ: обе системы были вскрыты из-за провалов в управлении ключами, а не из-за алгоритмических слабостей.
7. **Никогда не реализуйте криптографические алгоритмы самостоятельно**. Используйте проверенные, аудированные библиотеки.
8. **Планируйте криптографическую гибкость**. Криптографические стандарты эволюционируют; ваши системы должны быть способны эволюционировать вместе с ними.

На следующей неделе мы рассмотрим асимметричное шифрование, которое решает проблему распределения ключей — фундаментальную задачу симметричного шифрования. Как две стороны, которые никогда не встречались, могут установить общий секретный ключ? Ответ произвёл революцию в криптографии и сделал возможным безопасный интернет, на который мы полагаемся сегодня.

Вопросы для обсуждения

1. Почему большинство реальных криптографических отказов связаны с управлением ключами, а не со слабостями алгоритмов? Используйте случаи Энигмы и ВЕНОНЫ для обоснования ответа.
2. Как организациям следует планировать свою криптографическую стратегию для поддержания криптографической гибкости по мере развития стандартов?
3. Какую роль будет играть облегчённая криптография, такая как Ascon, по мере расширения Интернета вещей?
4. Если одноразовый блокнот обеспечивает совершенную секретность, почему весь мир использует AES? Что это говорит нам о разрыве между теоретической и практической безопасностью?
5. Советская фабрика, дублировавшая ключи одноразовых блокнотов, находилась под давлением военного времени и приняла прагматичное решение. В современных организациях аналогичные компромиссы возникают, когда команды безопасности испытывают давление ускорить выпуск продукта. Как организациям следует действовать в ситуациях, когда требования безопасности конфликтуют с операционной срочностью?
6. Энигма считалась невзламываемой немецким военным командованием, но была вскрыта. AES был опубликован более 25 лет назад и остаётся невзломанным. Что принципиально отличается в том, как AES был спроектирован и оценён, по сравнению с Энигмой?

Благодарю за внимание. До встречи на следующей неделе.

Контрольные вопросы

1. Объясните принципы Шеннона — перемешивание и рассеивание. Как они реализованы в современных шифрах?
2. В чём разница между вычислительной безопасностью и теоретико-информационной безопасностью? Какой тип обеспечивает AES (Advanced Encryption Standard, усовершенствованный стандарт шифрования) и какой — одноразовый блокнот?
3. Дайте определение перемешиванию, рассеиванию и лавинному эффекту. Объясните, как AES удовлетворяет каждому из этих свойств и как Энигма не смогла их обеспечить.
4. Сформулируйте принцип Керкгоффса. Почему Энигма его нарушала, и как современные шифры, такие как AES, ему соответствуют?

5. Какие условия должны быть выполнены, чтобы одноразовый блокнот обеспечивал совершенную секретность, и почему каждое условие необходимо?
6. Почему машина Энигма была вскрыта, несмотря на огромное пространство ключей? Какие конкретные криптографические принципы (перемешивание, рассеивание, лавинный эффект, принцип Керкгоффса) нарушала Энигма?
7. В случае ВЕНОНЫ какое криптографическое правило было нарушено и каковы были последствия?
8. В чём разница между блочным шифром и потоковым шифром? Приведите пример каждого.
9. Почему DES был выведен из эксплуатации, и какие основные слабости привели к его замене? Как Тройной DES пытался их устранить?
10. Опишите процесс выбора AES и объясните, почему был выбран Rijndael.
11. Назовите четыре операции одного раунда AES и объясните назначение каждой.
12. Сравните режимы работы ECB (Electronic Codebook, режим электронной кодовой книги) и CBC (Cipher Block Chaining, режим сцепления блоков шифротекста). Почему ECB никогда не следует использовать для шифрования данных?
13. Что может произойти, если зашифрованные данные не аутентифицированы? Приведите пример атаки, которую аутентификация предотвращает.
14. Что такое одноразовый номер (nonce) и почему его никогда нельзя использовать повторно с тем же ключом?
15. Почему WireGuard использует ChaCha20 вместо AES, и на каких устройствах ChaCha20 быстрее?
16. Что такое Ascon и для каких устройств он был разработан?
17. Что такое вывод ключей и почему пароль никогда не следует использовать напрямую в качестве ключа шифрования?
18. При хешировании паролей объясните, какие атаки возможны без соли, что предотвращает соль и какую дополнительную защиту обеспечивает перец. Почему комбинация индивидуальной соли и перца уровня приложения надёжнее, чем каждая мера по отдельности?
19. Почему ротация ключей важна и что произойдёт, если один и тот же ключ используется бессрочно?

Ключевые термины

- **AEAD**: Аутентифицированное шифрование с присоединёнными данными (Authenticated Encryption with Associated Data), обеспечивающее

конфиденциальность и целостность в одной операции

- **AES**: Усовершенствованный стандарт шифрования (Advanced Encryption Standard), симметричный блочный шифр с 128-битными блоками и ключами 128/192/256 бит
- **Ascon**: стандартизированный NIST алгоритм облегчённого аутентифицированного шифрования для ограниченных устройств
- **Аутентифицированное шифрование**: шифрование, обеспечивающее и конфиденциальность, и проверку целостности
- **Блочный шифр**: шифр, шифрующий блоки открытого текста фиксированного размера
- **CBC**: режим сцепления блоков шифротекста (Cipher Block Chaining), при котором каждый блок открытого текста XOR-ится с предыдущим блоком шифротекста
- **CCM**: режим счётчика с CBC-MAC (Counter with CBC-MAC), режим аутентифицированного шифрования, сочетающий CTR-шифрование с аутентификацией CBC-MAC
- **ChaCha20**: потоковый шифр, разработанный Дэниелом Бернштейном, используемый с Poly1305 для аутентифицированного шифрования
- **Перемешивание**: принцип Шеннона о создании сложной связи между ключом и шифротекстом
- **CTR**: режим счётчика (Counter mode), превращающий блочный шифр в потоковый
- **DES**: Стандарт шифрования данных (Data Encryption Standard), выведенный из эксплуатации блочный шифр с 56-битным ключом
- **Рассеивание**: принцип Шеннона о распространении влияния открытого текста по всему шифротексту
- **ECB**: режим электронной кодовой книги (Electronic Codebook), при котором каждый блок шифруется независимо
- **Энигма**: электромеханическая роторная шифровальная машина, использовавшаяся нацистской Германией, вскрытая союзными криптоаналитиками через эксплуатацию дефектов проектирования и эксплуатационных ошибок
- **GCM**: режим Галуа со счётчиком (Galois/Counter Mode), режим аутентифицированного шифрования, сочетающий CTR с GHASH
- **HKDF**: функция вывода ключей на основе HMAC (HMAC-based Key Derivation Function) для получения ключей из другого ключевого материала
- **Ротация ключей**: практика периодической замены криптографических ключей

- **Одноразовый номер (Nonce)**: число, используемое однократно, необходимое многим режимам шифрования для обеспечения уникальности шифротекстов
- **Одноразовый блокнот (OTP)**: единственная схема шифрования, доказанно обеспечивающая совершенную секретность, требующая истинно случайного ключа длиной не менее длины сообщения, используемого ровно один раз
- **PBKDF2**: функция вывода ключей из паролей (Password-Based Key Derivation Function 2), для получения ключей из паролей
- **Перец (Pepper)**: секрет уровня приложения, добавляемый при хешировании паролей, хранящийся отдельно от базы данных, обеспечивающий эшелонированную защиту при краже базы данных
- **Совершенная секретность**: теоретико-информационное свойство, при котором шифротекст не раскрывает никакой информации об открытом тексте; доказано Клодом Шенноном в 1949 году, что требует ключей длиной не менее длины сообщений
- **Потоковый шифр**: шифр, шифрующий открытый текст побитно или побайтно
- **Симметричное шифрование**: шифрование, при котором один и тот же ключ используется и для шифрования, и для расшифрования
- **Тройной DES (3DES)**: устаревший симметричный шифр, применяющий DES трижды с двумя или тремя разными ключами
- **ВЕНОНА**: американо-британский разведывательный проект (1943-1980), частично дешифровавший советские коммуникации путём эксплуатации повторного использования ключей одноразового блокнота
- **XChaCha20**: вариант ChaCha20 с расширенным одноразовым номером, поддерживающий 192-битные одноразовые номера для масштабного шифрования