

---



## 4.1 Что такое шифрование?

Для начала стоит дать определение открытому тексту. *Открытый текст* (plaintext) – это некая информация, которую можно брать и пользоваться. «Война и мир», картинка с котиком, пакет Python – все это может быть примером открытого текста. *Зашифровка* – это умышленное обратимое искажение открытого текста с целью скрыть

информацию от тех, кому ее видеть не следует. Результатом этого процесса является зашифрованный текст (ciphertext).

Обратный процесс, то есть преобразование шифротекста в открытый с применением ключа, называется *расшифровкой*<sup>1</sup>. Алгоритм зашифровки и расшифровки данных называется *шифром*. Для применения шифра требуется ключ. Ключ должны знать только те, у кого есть право доступа к информации.

Термин шифрование объединяет процессы зашифровки и расшифровки.

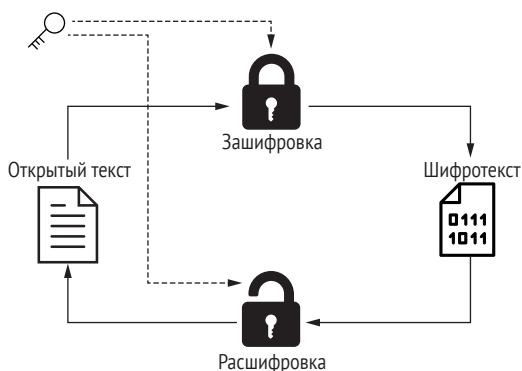


Рис. 4.1 Открытый текст передается на зашифровку и является результатом расшифровки. Шифротекст – это продукт зашифровки, к нему применяется расшифровка

Шифрование отвечает за неразглашение данных. При создании защищенной системы неразглашение является одним из фундаментальных принципов наравне с целостностью данных и проверкой их подлинности. По сравнению с другими азами у неразглашения очень простое определение: обеспечить секретность. В этой книге я делю ее на два вида:

- секретность личная;
- секретность групповая.

Рассмотрим примеры. Алиса хочет прочитать и записать конфиденциальную информацию. Другим лицам должно быть невозможно прочесть эти данные. Зашифровывая данные при записи и расшифровывая при чтении, Алиса может обеспечить личную секретность. Вспомним о проверенном приеме из первой главы: *шифрование хра-*

<sup>1</sup> Не следует путать *расшифровку* с *дешифровкой*. Дешифровкой является получение открытого текста из шифротекста без знания секретного ключа методами криптоанализа. При этом в английском языке термин *decryption* чаще всего обозначает расшифровку и достаточно редко дешифровку, отдельного понятия для последней там не устоялось. Стоит заметить, что русские понятия *шифрование* как процесс зашифровки-расшифровки и сам термин *зашифровка* тоже передаются единым английским словом *encryption*. – Прим. перев.

нимых и передаваемых данных. Получается, шифрование хранимых данных как раз и отвечает за личную секретность.

В другом случае Алиса хочет передать нечто конфиденциальное Бобу, получить – то же. Зашифровывая отправляемые данные и расшифровывая получаемые, они создадут условия для групповой секретности. Следовательно, шифрование передаваемых данных – в ответе за групповую секретность.

Далее в этой главе рассказывается, как с помощью Python и пакета сгустогарфу реализовать шифрование хранимых данных. Чтобы воспользоваться данным пакетом, сначала нам придется установить безопасный пакетный менеджер.

### 4.1.1 Управление пакетами

Для управления пакетами в этой книге используется Pipenv. Выбор пал на него из-за того, что в нем есть широкий функционал для обеспечения безопасности проекта. О некоторых достоинствах этого менеджера будет написано в главе 13.

**ПРИМЕЧАНИЕ** Существует множество пакетных менеджеров для Python. Вам не обязательно использовать тот же, что использую я; примеры кода будут работать безотносительно выбранного менеджера. Можете положиться как на `pip`, так и на `venv` вместо Pipenv, но вы останетесь без некоторых его защитных возможностей.

Чтобы установить Pipenv, выберите среди команд подходящую для вашей операционной системы. Не стоит ставить Pipenv на macOS через Homebrew, использование LinuxBrew также не рекомендуется.

<code>\$ sudo apt install pipenv</code>	← Debian версии Buster и новее
<code>\$ sudo dnf install pipenv</code>	← Fedora
<code>\$ pkg install py36-pipenv</code>	← FreeBSD
<code>\$ pip install --user pipenv</code>	← Остальные ОС

Затем введите эту команду. Она создаст два файла в текущей директории: `Pipfile` и `Pipfile.lock`. Они нужны Pipenv, чтобы отслеживать зависимости вашего проекта.

```
$ pipenv install
```

Кроме этих файлов, команда выше еще создаст виртуальное окружение. Это изолированная самодостаточная среда для выполнения проекта на Python. Каждое такое окружение довольствуется своим собственным интерпретатором Python, набором библиотек и скриптов. Если каждый проект заключен в подобной среде, то они не смогут негативно влиять на работу друг друга. Эта команда запустит только что созданное виртуальное окружение:

```
$ pipenv shell
```

**ВНИМАНИЕ!** Сделайте себе одолжение – запускайте все команды из этой книги внутри консоли только что созданной виртуальной среды. Благодаря этому у вас не будет проблем с нахождением зависимостей. Кроме того, установленные вами для этого проекта зависимости не вступят в конфликт с зависимостями других проектов.

Вы должны запускать команды из примеров кода внутри виртуального окружения, как и следует делать в большинстве проектов на Python. В следующем разделе вы установите в виртуальной среде первую из многих зависимостей – пакет `cryptography`. Он исчерпывающе покрывает нужды программиста при реализации шифрования.

## 4.2 *Пакет cryptography*

В отличие от некоторых других языков программирования, в Python отсутствуют встроенные средства для шифрования. Несколько библиотек с открытым кодом восполняют этот пробел. Самыми популярными пакетами для криптографии являются `cryptography` и `pycryptodome`. В книге используется исключительно `cryptography`, так как в нем меньше возможностей «выстрелить себе в ногу». В данном разделе описан самый необходимый его функционал.

Чтобы установить пакет `cryptography`, введите в виртуальном окружении:

```
$ pipenv install cryptography
```

По умолчанию за кулисами пакета трудится библиотека OpenSSL, имеющая открытый код. В ней реализованы сетевые протоколы, отвечающие за безопасность, и криптографические функции для широкого круга задач. В основном библиотека написана на C. Она также находится под капотом множества других библиотек для всевозможных языков программирования.

Авторы пакета разделили доступные в нем возможности на две категории:

- «взрывчатые вещества», нетривиальные низкоуровневые инструменты;
- «готовые рецепты», высокоуровневые и несложные в использовании.

### 4.2.1 «Взрывчатые вещества»

Низкоуровневый и сложный в использовании API, скрытый за `cryptography.hazmat`, известен под названием «взрывчатые вещества» (`hazardous materials layer`). Хорошенько подумайте, прежде чем та-

щить его в боевую систему. Документация (<https://cryptography.io/en/latest/hazmat/primitives/>) гласит: «Используйте все это, только если вы отдаете себе полный, стопроцентный отчет в том, что вы делаете. Это минное поле кишмя кишит драконами и динозаврами». Обращение с этими инструментами требует досконального знания криптографии. Малейшая ошибка – и ваша система под угрозой.

Уважительных причин лезть сюда практически нет. Разве что:

- нужно шифровать файлы, которые не помещаются в оперативной памяти;
- нужно использовать редкий шифр;
- в какой-нибудь книжке объясняются азы через низкоуровневый API.

## 4.2.2 «Готовые рецепты»

Высокоуровневый и простой в использовании API называется «готовые рецепты». Цитирую документацию (<https://cryptography.io/en/latest/>): «Стоит использовать “готовые рецепты” всегда и везде и прибегать к “взрывчатым веществам” только в случае крайней нужды». Большинству программистов на Python для шифрования будет достаточно готовых рецептов.

Один из готовых рецептов реализует метод симметричного шифрования под названием *fernet*. Его нормативная документация описывает протокол для зашифрованного взаимодействия, стойкий к постороннему вмешательству. Его воплощает класс `Fernet`, который находится в `cryptography.fernet`.

В классе `Fernet` есть все, что вам, как правило, потребуется для шифрования данных. Метод `Fernet.generate_key()` создает ключ длиной 32 случайных байта, который требуется конструктору класса в качестве аргумента:

```
>>> from cryptography.fernet import Fernet
>>>
>>> key = Fernet.generate_key()
>>> fernet = Fernet(key)
```

За `cryptography.fernet` и скрывается простой API

Под капотом `Fernet` делит переданный ключ на два 128-битных. Один используется для шифрования, а второй – для проверки подлинности, о которой говорилось в предыдущей главе.

Метод `Fernet.encrypt` не только зашифровывает открытый текст, он также высчитывает хеш от шифротекста функцией HMAC-SHA256. То есть шифротекст для хеш-функции является сообщением. Шифрованный текст и хеш возвращаются внутри объекта под названием *fernet token*:

```
>>> token = fernet.encrypt(b'plaintext')
```

Зашифровывает открытый текст, хеширует шифротекст

На рис. 4.2 изображено, как из зашифрованного текста и хеша формируется токен. Ключи шифрования и хеширования не показаны для упрощения.

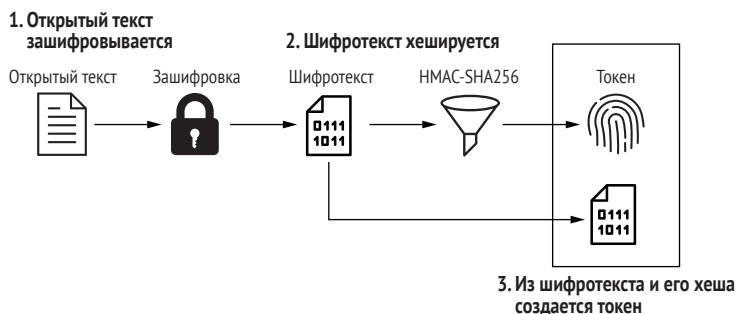


Рис. 4.2 Fernet не только зашифровывает, но и хеширует

Методу `Fernet.encrypt` противопоставлен метод `Fernet.decrypt`. Он извлекает шифротекст из токена и проверяет его подлинность с помощью HMAC-SHA256. Если заново посчитанный хеш не совпадает с хешем внутри токена, то будет брошено исключение `InvalidToken`. Если хеши совпадают, метод расшифровывает сообщение и возвращает его:

```
>>> fernet.decrypt(token)  ← Проверкает подлинность
b'plaintext'               ← и расшифровывает шифротекст
```

На рис. 4.3 показано, как метод `Fernet.decrypt` обращается с токеном. Как и на предыдущем рисунке, ключи не изображены.



Рис. 4.3 Fernet проверяет подлинность шифротекста и только потом расшифровывает его

Может возникнуть вопрос: а разве одного только неразглашения недостаточно, зачем еще проверять и подлинность? Ценность неразглашения раскрывается в полной мере при использовании вместе с проверкой подлинности. Допустим, Алиса хочет обеспечить

личную секретность. Все, что она пишет, – зашифровывает; все, что она читает, – расшифровывает. Храня ключ в секрете, Алиса может быть уверена, что только она может расшифровать зашифрованный текст. Но откуда ей знать, она ли автор этого шифротекста? Проверка подлинности – это дополнительный уровень обороны от Мэллори, которой может быть выгодно подменить шифротекст.

Допустим, Алисе и Бобу требуется групповая секретность. И та, и другой шифруют свое общение. Ключ шифрования находится только у них, следовательно, Ева не сможет обозревать их переписку. Но этого недостаточно, для того чтобы Алиса была уверена, что отправитель сообщений – именно Боб, и наоборот. Только проверка подлинности может дать такую гарантию.

Кроме того, сам токен `fernet` устроен так, чтобы минимизировать соблазн сделать с ним что-нибудь небезопасное. Каждый токен – обыкновенный массив байтов, а не какой-нибудь класс `FernetToken` со свойствами для шифротекста и хеша. Если прямо очень нужно, извлечь из массива хеш и зашифрованный текст можно, но достаточно неопытным способом. Строение токенов нарочно не поощряет написание потенциально содержащего ошибки кода, как то свой собственный расшифровщик и контроль подлинности, и удерживает от расшифровки без предварительной проверки подлинности. Такая реализация токена предохраняет от нарушения проверенного приема «не изобретай свое шифрование», о котором говорилось в первой главе. `Fernet` создан, чтобы его просто было использовать по-безопасному и сложно было бы применить не так.

Объект класса `Fernet` может расшифровать токен, созданный этим же объектом либо другим, но хранящим тот же ключ. Экземпляр класса можно без проблем уничтожить, но обязательно нужно позаботиться о сохранности ключа. Без него невозможно будет восстановить открытый текст. В следующем разделе поговорим о смене секретных ключей с помощью `MultiFernet`, побратима `Fernet`.

### 4.2.3 Смена ключа

Смена ключа нужна для изъятия из оборота старого ключа и замены его на новый. Чтобы сделать это, требуется расшифровать весь шифротекст, зашифрованный старым ключом, и зашифровать новым. Смена может требоваться по многим причинам. Разглашенный ключ должен быть изменен немедленно. Иногда ключи меняют, когда некто, прежде работавший в команде, покидает ее. Регулярная их смена может снизить урон, который нанесет разглашение ключа, но на вероятность разглашения повлиять не получится.

Класс `MultiFernet` используется для миграции с одного ключа на другой. Для этого создаются два объекта `Fernet`: один содержит в себе старый ключ, а другой – новый. Оба этих объекта передаются конструктору класса `MultiFernet`. Его метод `rotate` расшифровывает токен старым ключом и зашифровывает новым. Как только все

токены были зашифрованы новым ключом, можно спокойно избавляться от старого.

#### Листинг 4.1 Смена ключа с помощью MultiFernet

```
from cryptography.fernet import Fernet, MultiFernet

old_key = read_key_from_somewhere_safe() ← Читаем ключ из безопасного места
old_fernet = Fernet(old_key)

new_key = Fernet.generate_key() ← Создаем новый
new_fernet = Fernet(new_key)

multi_fernet = MultiFernet([new_fernet, old_fernet])
old_tokens = read_tokens_from_somewhere_safe()
new_tokens = [multi_fernet.rotate(t) for t in old_tokens]

replace_old_tokens(new_tokens)
replace_old_key_with_new_key(new_key)
del old_key

for new_token in new_tokens:
    plaintext = new_fernet.decrypt(new_token)
```

Расшифровываем старым ключом, зашифровываем новым

Кладем на место старых токенов и ключа новые

Теперь для расшифровки нужен новый ключ

От того, как ключ используется, зависит категория, к которой причисляется алгоритм шифрования. Следующий раздел расскажет о категории, в которую попадает Fernet.

## 4.3 Симметричное шифрование

Если алгоритму шифрования требуется один и тот же ключ как для зашифровки открытого текста, так и для его расшифровки – прямо как знакомому нам Fernet, – такое шифрование называется *симметричным*. Алгоритмы симметричного шифрования делятся на две подкатегории: блочные шифры и потоковые.

### 4.3.1 Блочные шифры

*Блочные шифры* зашифровывают открытый текст в последовательность блоков шифротекста одинаковой длины. Открытый текст делится на блоки, над которыми и проводится зашифровка. Размер блока зависит от алгоритма. Обычно чем он больше, тем более стойким считается шифрование. На рис. 4.4 три блока открытого текста зашифрованы в три блока шифрованного.

Алгоритмов симметричного шифрования достаточно много, и поначалу выбор между ними может показаться трудным. Какой безопаснее, какой быстрее? На самом деле это простые вопросы, чуть ниже вы прочтете ответы. Популярными блочными шифрами являются:

- Triple DES;
- Blowfish;
- Twofish;
- Advanced Encryption Standard.



Рис. 4.4 Если блочному шифру подать на вход  $N$  блоков открытого текста, то на выходе получим  $N$  блоков шифротекста

## TRIPLE DES

Triple DES (3DES) создан на основе Data Encryption Standard (DES). Как подсказывает имя, под капотом блок шифруется алгоритмом DES в три прогона. Поэтому этот шифр считается медленным. Размер блока – 64 бита, длина ключа – 56, 112 либо 168 бит.

**ВНИМАНИЕ!** Национальный институт стандартов и технологий, а также OpenSSL не рекомендуют использовать 3DES, он объявлен устаревшим. С публикацией института можно ознакомиться по ссылке <https://mng.bz/pJoG>.

## BLOWFISH

Blowish был разработан Брюсом Шнайером (Bruce Schneier) в начале 90-х. Размер блока – 64 бита, ключ любой длины от 32 до 448 бит. Шифр был первым незапатентованным и не требовал денежных отчислений при использовании – это обеспечило ему популярность.

**ВНИМАНИЕ!** Blowish потерял признание, когда в 2016 году из-за размера блока оказался уязвим к атаке SWEET32. Не применяйте этот шифр. Даже его создатель рекомендует отказаться от него в пользу Twofish.

## TWOFISH

Twofish был разработан в конце 90-х на замену Blowfish. Размер блока – 128 бит, длина ключа – 128, 192 либо 256 бит. Шифр получил признание криптографов, но популярности предшественника не достиг.

В 2000 году он стал одним из пяти финалистов трехлетнего конкурса Advanced Encryption Standard. Можно спокойно использовать Blowfish, но почему бы не поступить как все и не предпочесть победителя конкурса?

## ADVANCED ENCRYPTION STANDARD

*Rijndael* (произносится «рейндаел») – алгоритм шифрования, который победил свыше десятка других шифров в конкурсе Advanced Encryption Standard. После этого в 2001 году Национальный институт стандартов и технологий выпустил на него нормативный документ. Вряд ли вы слышали об этом шифре прежде, хотя он служит вам каждый день. Все потому, что его стали называть *Advanced Encryption Standard* (AES) по имени конкурса.

Единственный алгоритм симметричного шифрования, о котором нужно знать рядовому программисту, – именно AES. Размер блока – 128 бит, длина ключа – 128, 192 либо 256 бит. Среди алгоритмов симметричного шифрования это образец для подражания. У него широкий и внушающий послужной список. AES применяется в сетевых протоколах, например HTTPS, а также в алгоритмах сжатия, внутри файловых систем, при вычислении хешей и для установления виртуальных частных сетей (VPN). У какого еще шифра есть поддержка на уровне команд центрального процессора? Как ни пытайтесь, у вас не получится создать систему, в которой не используется AES.

И конечно же, под капотом у Fernet трудится AES. Для рядовых задач этот шифр – самый оптимальный выбор. Не играйте с огнем и забудьте о других блочных шифрах. Следующий раздел расскажет о потоковых шифрах.

### 4.3.2 Потоковые шифры

*Потоковые шифры* не делят открытый текст на блоки. Вместо этого они обрабатывают его как поток несвязанных байтов: один вошел, один вышел. Эти шифры подходят для обработки непрерывного потока данных, либо когда размер открытого текста просто неизвестен. Поэтому они часто применяются в сетевых протоколах.

Когда открытый текст очень мал, потоковые шифры показывают себя лучше блочных. Допустим, вы применяете блочный шифр, размер открытого текста 120 бит, размер блока 128 бит. Шифру придется дополнить 8 бит для создания блока шифротекста, как будто длина исходного текста делится на 128 нацело. А теперь пусть открытый текст будет размером 8 бит, это уже 120 дополнительных бит. Получается, больше 90 % шифротекста будут ничего не значащими данными. Потоковые шифры лишены этого недостатка: им не нужно дополнять открытый текст, ведь им не требуются равные блоки оного.

RC4 и ChaCha – примеры потоковых шифров. Пока в RC4 не вскрылось полдесятка уязвимостей, он широко применялся в сетевых

протоколах. С этим шифром давно попрощались, использовать его ни в коем случае нельзя. ChaCha же считается защищенным и весьма-весьма быстрым. Когда в шестой главе мы будем говорить о TLS, безопасном сетевом протоколе, этот шифр нам еще встретится.

Потоковые шифры, несмотря на скорость и эффективность, не востребованы настолько, насколько блочные. Увы, успешное вмешательство со стороны куда вероятно в шифротекст, порожденный потоковым шифром, нежели блочным. В зависимости от режима шифрования блочные шифры могут имитировать работу потоковых. О режимах шифрования – следующий раздел.

### 4.3.3 Режимы шифрования

Алгоритмы симметричного шифрования могут работать в разных режимах. У каждого из них есть свои плюсы и минусы. Когда речь идет о применении симметричного шифрования, то обычно это не разговор о том, потоковые или блочные, тот алгоритм или другой. Обычно это обсуждение того, какой режим выбрать для алгоритма AES.

#### РЕЖИМ ЭЛЕКТРОННОЙ КОДОВОЙ КНИГИ

Самый незатейливый метод применения блочного шифра – это *режим электронной кодовой книги* (electronic codebook mode – ECB), по ГОСТ – *режим простой замены*. Чуть ниже расположен пример кода, зашифровывающий данные алгоритмом AES в этом режиме. С помощью низкоуровневых возможностей пакета cryptography определяется шифр с ключом длиной 128 бит. Открытый текст передается через метод update. Для упрощения блок открытого текста только один без дополнения незначащими данными:

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.ciphers import (
...     Cipher, algorithms, modes)
>>>
>>> key = b'key must be 128, 196 or 256 bits'
>>>
>>> cipher = Cipher(
...     algorithms.AES(key),
...     modes.ECB(),
...     backend=default_backend())
>>> encryptor = cipher.encryptor()
>>>
>>> plaintext = b'block size = 128'
>>> encryptor.update(plaintext) + encryptor.finalize()
b'G\xf2\xe2J]a;\x0e\xc5\xd6\x1057D\xa9\x88'
```

Будет применен AES в режиме ECB

Под капотом будет OpenSSL

Один блок открытого текста

Один блок шифротекста

Шифрование через режим электронной кодовой книги отличается удивительно низкой стойкостью, зато простота его применения

отлично подходит для наглядных примеров. Этот режим небезопасен потому, что одинаковые блоки открытого текста он превращает в одинаковые блоки шифрованного. Благодаря этому с ним легко разобраться новичку, но и злоумышленнику тоже становится просто угадать структуру открытого текста по шифротексту.

На рис. 4.5 показан классический пример уязвимости режима. Слева – обыкновенная картинка, справа – она же, но зашифрованная<sup>1</sup>.



Рис. 4.5 При использовании режима ECB структура открытого текста воспроизводится в структуре шифротекста

Режим электронной кодовой книги не просто раскрывает структуру внутри отдельно взятого открытого текста, но еще и обнажает совпадения между разными текстами. Допустим, Алисе надо зашифровать некий набор открытых текстов. Она использует режим ECB, руководствуясь ложной предпосылкой: этот метод якобы безопасен, если отдельно взятый открытый текст не имеет различимой структуры. Мэллори удастся заполучить набор шифротекстов. Она их исследует и находит совпадающие между собой. Почему так вышло? В отличие от Алисы, Мэллори в курсе, что режим простой замены зашифровывает одинаковые открытые тексты в одинаковые шифротексты.

**ВНИМАНИЕ!** Никогда не шифруйте данные в режиме ECB на боевых системах. То, что используется безопасный алгоритм AES, вообще не спасает ситуацию. Режим электронной кодовой книги и безопасность – вещи несовместимые.

Если взломщик заполучил доступ к вашим шифротекстам, ему должно быть невозможно понять по ним хоть что-нибудь об открытом тексте. Далее описывается хороший режим шифрования, который скрадывает структуру как отдельного текста, так и совпадения между ними.

<sup>1</sup> Источник изображения слева: <https://en.wikipedia.org/wiki/File:Tux.jpg>. © Larry Ewing, lewing@isc.tamu.edu и The GIMP. Источник изображения справа: [https://en.wikipedia.org/wiki/File:Tux\\_ecb.jpg](https://en.wikipedia.org/wiki/File:Tux_ecb.jpg).

## РЕЖИМ СЦЕПЛЕНИЯ БЛОКОВ ШИФРОТЕКСТА

Режим сцепления блоков шифротекста (cipher block chaining – CBC) не страдает болезнью предыдущего режима. Достигается это тем, что любое изменение в блоке влияет на шифротекст блоков последующих. На рис. 4.6 показано, что структура открытого текста не видна в шифротексте<sup>1</sup>.



Рис. 4.6 При использовании режима CBC структура открытого текста не воспроизводится в структуре шифротекста

Кроме того, зашифровка в этом режиме одинаковых открытых текстов дает разные шифротексты. Это достигается с помощью *вектора инициализации* (initialization vector – IV). Он подается шифру на вход вместе с текстом и ключом. Вектор для режима сцепления блоков шифротекста должен быть случайным 128-битным числом, он должен быть использован только однократно.

Этот пример кода зашифровывает два одинаковых текста из двух одинаковых блоков алгоритмом AES в режиме CBC. Для каждого текста вектор инициализации генерируется заново. Обратите внимание: шифротексты уникальны, блоки внутри шифротекста тоже не равны:

```
>>> import secrets
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.ciphers import (
...     Cipher, algorithms, modes)
>>>
>>> key = b'key must be 128, 196 or 256 bits'
>>>
>>> def encrypt(data):
...     iv = secrets.token_bytes(16)  ← 16 случайных байт
...     cipher = Cipher(
...         algorithms.AES(key),  ← Будет применен AES в режиме CBC
...         modes.CBC(iv),
...         backend=default_backend())
...     encryptor = cipher.encryptor()
```

<sup>1</sup> Источник изображения слева: <https://en.wikipedia.org/wiki/File:Tux.jpg>. © Larry Ewing, lewing@isc.tamu.edu и The GIMP. Источник изображения справа: [https://en.wikipedia.org/wiki/File:Tux\\_secure.jpg](https://en.wikipedia.org/wiki/File:Tux_secure.jpg).

```

...     return encryptor.update(data) + encryptor.finalize()
...
>>> plaintext = b'the same message' * 2
>>> x = encrypt(plaintext)
>>> y = encrypt(plaintext)
>>>
>>> x[:16] == x[16:]
False
>>> x == y
False

```

← Два одинаковых блока открытого текста  
 Зашифровка одинаковых текстов  
 Два одинаковых блока открытого текста стали двумя разными блоками шифротекста  
 Два одинаковых открытых текста стали двумя разными шифротекстами

Использованный вектор требуется и при последующей расшифровке, наравне с шифротекстом и ключом. Следовательно, IV требуется сохранить. Без него открытый текст будет безвозвратно потерян.

Fernet применяет AES в режиме CBC и берет заботу о векторе на себя. Он будет создан при зашифровке текста, положен в токен вместе с шифротекстом и хешем и вытащен оттуда перед расшифровкой.

**ВНИМАНИЕ!** Некоторые программисты прячут вектор инициализации, будто это ключ. Ключ предназначен для зашифровки одного и более сообщений, вектор же – для одного и только одного сообщения. Ключ должен храниться в секрете, вектор же обычно кладут рядом с шифротекстом безо всяких премудростей. Если злоумышленник стащил ваши шифротексты, считайте, что и векторы тоже. Без ключа они ему все равно ничего не дадут.

AES может работать и в других режимах шифрования. Один из них, счетчик с аутентификацией Галуа (Galois/counter mode – GCM), позволяет блочному шифру имитировать работу потокового. Мы с ним еще встретимся в шестой главе.

## Итоги

- Шифрование обеспечивает неразглашение.
- Fernet – безопасный и простой способ симметричного шифрования и проверки подлинности данных.
- MultiFernet облегчает смену ключей.
- В алгоритмах симметричного шифрования применяется один и тот же ключ для зашифровки и расшифровки.
- Если симметричное шифрование, то AES.