Проблема сохранения конфиденциальности данных пользователей очень важна в современном мире. Данные должны быть защищены всеми возможными способами. От нас этого требует не только мораль, но и закон. И как вы хотите обеспечить их безопасность, пользуясь алгоритмами шифрования, изобретёнными в прошлом веке? Очевидно, что такое невозможно, так как эти шифры имеют по несколько способов взлома. И доступ к самому дорогому, что у вас есть, будет открыт любому желающему. Такое отношение к пользователям безнравственно и незаконно.

Я сам с этим столкнулся. У меня в разработке имеется небольшой самописный коллективный блог. Для того чтобы предложить свою запись на общую ленту, пользователь должен зарегистрироваться, доверить мне свои личные данные. Эти данные обязательно надо защищать, тогда я решил поэкспериментировать с защитой. Поразмыслив, я понял, что в образовательных целях можно написать свою реализацию одного из алгоритмов.

Моя работа будет считаться выполненной, если в её ходе будет создана эффективная и надёжная реализация алгоритма шифрования.

Я использую элементарные клеточные автоматы по нескольким причинам:

* Это просто. Алгоритм работы элементарных клеточных автоматов прост для понимания и реализации. Его можно объяснить на пальцах.
* Это быстро. Вы не успеете моргнуть глазом, как символ будет зашифрован. Буквально.

В самом начале проекта я и не собирался ничего шифровать. Мне просто было приятно смотреть на эволюцию клеточных автоматов. Один их вид завораживает и заставляет вглядываться в детали замысловатого узора. Мой симулятор был разработан как для PC, так и для Android. Его написание я рассматривал как тренировку своих навыков в программировании.

Возможность использования клеточных автоматов в алгоритмах шифрования была открыта для меня недавно, хотя данный алгоритм уже существовал. То есть на первых порах я незаметно для себя создавал велосипед, пытаясь создать совершенно новый метод шифровки. Как он работает? На вход поступает строка символов в кодировке Юникод и номер клеточного автомата. Входная строка разбивается на символы, почему шифр и называется блочным. Коды символов переводятся в двоичную систему счисления и передаются на вход элементарному клеточному автомату. Этими двоичными разрядами инициализируется поле при создании. Потом мы просчитываем следующее поколение и переводим его в десятичную систему счисления, представив, что это просто двоичное число. В итоге у нас получается код нового символа в кодировке Юникод. Преобразованием кода в символ заботится вызывающая сторона. При этом, не пользуясь словарём, невозможно получить исходный символ, из-за чего шифрование получается необратимым. Но не беспокойтесь, это не проблема. Такие алгоритмы активно используются в наше время, например при авторизации пользователя, где сравниваются не исходные пароли, а они же в зашифрованном виде.

Но и этот алгоритм не идеален – в нём присутствует огромное количество коллизий – совпадений зашифрованных символов, когда исходные символы не равны. Обратимся к примеру. Используя правило 110, зашифруем символ L. Получаем Ü. Шифруем t – тоже Ü. И что дальше, почему это плохо? А вот сами посмотрите: Lime = time, Look = took, Light = tight. То есть зашифрованные аналоги этих слов будут равны! Давайте вспомним пример с паролями. Представляете, какое лицо будет у владельца аккаунта, если система хранения паролей потакает взломщикам, позволяя совершать ошибки? Но на деле воспользоваться этой уязвимостью не так просто – хакер не знает, какой номер клеточного автомата используется. А на перебор всех номеров уходит достаточное время, при том, что длинна пароля так же неизвестна. За время перебора атака будет вычислена и остановлена.

Что делать с многобайтовыми кодировками? В C++ нет стандартного типа, который обязательно поддерживал бы нужную нам кодировку. Размер wchar\_t не стандартизирован, поэтому и в нём уверенности никакой нет. Поэтому алгоритм получает на вход не сам символ, а его код. Но каков диапазон значений этого кода? Сколько байтов отводится под символ? Ответы на этот вопрос известны только конечному пользователю – программисту. Тогда я пришёл к выводу, что не стоит использовать какой-то определённый целочисленный тип. Подключать сторонние библиотеки нежелательно, это скажется на производительности и переносимости. Поэтому нужно найти изящное решение, не противоречащее стандарту C++. Этим решением для меня является шаблон класса. Ознакомившись со стендом, вы узнаете примерную схему программы. Так вот, я решил, что класс Field стоит сделать шаблонным. Тогда мы решим описанную проблему, так как пользователь сам будет определять нужный ему диапазон значений, указывая целочисленный тип самостоятельно. А вдруг он укажет тип float или double? Не беда, так как алгоритм использует операцию побитового И, определённую только для целочисленных типов.

Если мои поиски были достаточно усердны, то можно утверждать, что моя программа уникальна. А из этого следует, что сравнить результаты тестов мне не с чем. Это не есть хорошо, но всё-таки тесты должны быть проведены. Вот некоторые факты: 1 символ –> за 80 тиков, все символы юникод -> 650 000 тиков, все символы через все правила -> 130 000 000. То есть вы не успеете моргнуть глазом, как все 65536 символов будут зашифрованы. Поверьте, это быстро. Все результаты усреднены для 10 запусков и округлены.

Над программой ещё предстоит много работы, перед её реальным использованием. К примеру, есть много способов улучшить производительность, некоторые из них приведены на моём стенде. А для использования необходимо получить сертификат ФСБ. Конечно, до сертификата ещё очень далеко, но замечательные результаты программа выдаёт уже сейчас. Если вы знаете, как дополнить программу, обращайтесь в github.com/Anfrew15381