Лабораторная работа № 5

Криптографические библиотеки

Цель работы

Познакомиться с существующими криптографическими библиотеками. Научиться использовать сторонние криптографические библиотеки при разработке собственных приложений.

Методические указания к работе

Найти в сети Интернет какую-либо криптографическую библиотеку (либо использовать криптографическую библиотеку, встроенную в выбранный язык программирования), позволяющую выполнять следующие четыре типа криптопреобразований:

- шифрование и дешифрование симметричными алгоритмами;
- шифрование и дешифрование асимметричными алгоритмами;
- вычисление и проверку электронной цифровой подписи;
- хэширование.

Изучить способы работы с выбранной библиотекой и реализовать программу, выполняющую указанные в задании действия.

Библиотека Crypto++

Одной из свободных и, пожалуй, самой функциональной и распространённой библиотекой для языка C++ является пакет ***Crypto++*** (другое название – **`CryptoPP`**) [1], разрабатываемый открытым сообществом и распространяемый под свободной лицензией **`Crypto++ License`** [2]. Его отличительными особенностями по сравнению с другими подобными библиотеками являются кроссплатформенность, богатые функциональные возможности и высокая скорость работы.

В состав библиотеки входят реализации таких классов алгоритмов, как:

- высокопроизводительные потоковые алгоритмы;
- симметричные алгоритмы;
- хэш-функции;
- алгоритмы с публичным ключом (асимметричные);
- генераторы псевдослучайных чисел;
- другие алгоритмы.

При разработке библиотеки сообщество максимально широко использовало возможности языка C++ и, в конечном итоге, получилась лаконичная, простая для использования и быстрая библиотека, которая может удовлетворить потребность разработчика в очень широком классе алгоритмов.

Данная библиотека реализует свой функционал, опираясь на принципы объектно-ориентированного программирования, богато эксплуатируемые везде, где это только возможно. Эта архитектурная особенность позволяет упростить разработку с использованием данной библиотеки, не уделяя пристального внимания (со стороны разработчика конечного программного обеспечения) особенностям алгоритмов, но оставляя возможность варьировать используемый алгоритм.

Идеология и принципы работы с библиотекой

Работать с данной библиотекой будет заметно проще тем, кто когда-либо использовал возможности таких пакетов, как **`Boost`**, или использовал из возможностей стандартной библиотеки **`std`** нечто большее, чем **`std::vector<int>`**.

Если рассматривать библиотеку в целом, то она представляет собой несколько базовых принципов и классов, от которых так или иначе унаследованы (в том числе и множественно) практически все остальные классы библиотеки. Сами классы и их иерархия построены таким образом, чтобы как можно меньше ограничивать разработчика в выборе способов комбинирования последовательности алгоритмов. Пример использования классов библиотеки **Crypto++**:

```
FileSource
  (inFileName,
   true,
   new HexDecoder
     (new StreamTransformationFilter
        (Decryptor,
         new HexEncoder (new FileSink (outFileName))
        )
     )
  );
```

С первого взгляда на этот «простой» код даже опытный программист, пишущий на C++, может прийти в некоторое замешательство, однако, немного разобравшись, поймёт, что архитектура библиотеки предоставляет поистине огромные возможности.

Если рассмотреть подробнее приведённый пример, можно понять, что этими лаконичными строчками кода выполняются следующие действия:

- берётся содержимое некоторого файла с именем **inFileName**;
- приводится из текстовой записи шестнадцатеричного вида к бинарному формату с помощью некоторого «потокового» алгоритма преобразования, задаваемого объектом **Decryptor**;
- преобразовывается обратно в текстовое шестнадцатеричное представление;
- сохраняется в файле с именем **outFileName**.

Казалось бы, выполнено большое количество действий, однако вместо конструирования нескольких статических объектов, чтения из файла, контроля над дешифрованием, переводов из/в бинарный вид, были сконструированы всего лишь один статический объект и пара динамических, которые и выполнили всю работу, при этом полностью проконтролировав все переданные им объекты/параметры на правильность.

В контексте библиотеки это называется ***Pipelining*** [3]. ***Pipelining*** – это парадигма, которая позволяет направить поток данных через набор фильтров, которые некоторым образом обрабатывают и/или модифицируют данные, от источника данных к приёмнику. Наиболее наглядным объяснением такого принципа является идеология конвейеров в Unix-подобных операционных системах, откуда она и была позаимствована разработчиками библиотеки.

Например, официальное вики приводит следующую аналогию (см. рисунок 1):

Рисунок 1 – Графическая аналогия для парадигмы Pipelining

Конвейер команд в Unix:

```
cat filename | base64 > filename.b64
```

Набор фильтров в **Crypto++**:

```
FileSource (filename,
            new Base64Encoder (new FileSink (filename + ".b64")));
```

Данная парадигма может быть применена в тесной связи с ещё одним механизмом, заложенным в библиотеку, – **ChannelSwitch**. Он реализован в одноимённом классе, который добавляет поддержку концепции *Read-Once Write-Many* (см. рисунок 2):

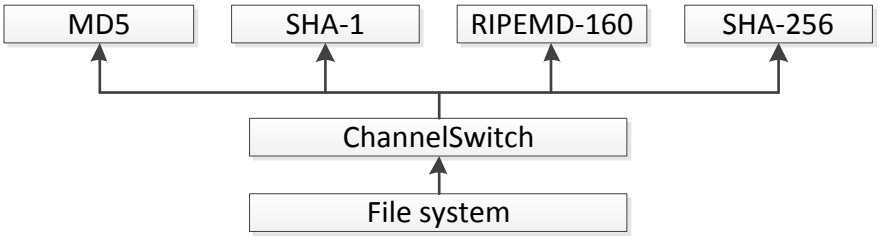

Рисунок 2 – Механизм ChannelSwitch

Тут же стоит заметить, что представленные куски кода не должны вводить в заблуждение: динамическое создание объектов – не единственный подход, возможность статического конструирования всего того, что было сделано динамически, всё так же остаётся, однако со статическим подходом следует обращаться более аккуратно, чем с динамическим. Казалось бы, нонсенс: обычно контролировать приходится именно динамические объекты, ведь в противном случае не избежать утечек памяти. Однако данная библиотека разрабатывалась именно для динамической схемы связывания алгоритмов, и в тот момент, когда будет вызван деструктор «родительского» статического объекта, для каждого «дочернего» объекта рекурсивно будут вызваны деструкторы.

Пример использования статических и динамических объектов. Статические объекты:

```
std::string digest, hex_digest;
...
HexEncoder encoder;
encoder.Attach (new StringSink (hex_digest));
encoder.Put ((byte *) digest.c_str (), digest.length ());
encoder.MessageEnd ();
```

Динамические объекты:

```
std::string digest, hex_digest;
...
StringSource (digest,
              true,
              new HexEncoder (new StringSink (hex_digest))
             );
```

Какой бы вариант кода ни был использован, результат будет одним и тем же: данные из строки **`digest`** после приведения к шестнадцатеричному текстовому представлению будут сохранены в строку **`hex_digest`**.

Разобрав несколько подобных примеров кода и поняв иерархическую модель библиотеки, можно в достаточной степени лаконично и эффективно использовать возможности **`Crypto++`**.

Помимо повсеместного эксплуатирования принципов инкапсуляции и полиморфизма, в реализации библиотеки не обойдён стороной и такой удобный механизм языка C++, как шаблоны. Для всех классов алгоритмов, параметры которых каким-либо образом можно варьировать, задание параметров реализовано именно через механизм шаблонов. Например, можно рассмотреть несколько алгоритмов блочного шифрования:

```
CFB_Mode<DES_EDE3>::Encryption Encryptor;
CFB_Mode<AES>::Encryption Encryptor;
CFB_Mode<Blowfish>::Encryption Encryptor;
```

Таким образом, было определено три алгоритма шифрования (**`3DES`**, **`AES`**, **`Blowfish`**), каждый из которых работает в режиме **`Cipher Feedback`**.

Данный пример не совсем хорош в том плане, что логичнее было бы варьировать режимы шифрования для каждого алгоритма и получать что-то вроде (*внимание, пример неправильный*):

```
AES<CFB_Mode>::Encryption Encryptor;
```

т.е. варьировать режимы работы для каждого алгоритма, но не все то, что «логичнее», более эффективно. Поскольку алгоритмов много, а режимы у всех одни и те же, разработчики библиотеки «обернули» именно алгоритмы в режимы, а не наоборот.

Таким образом, у библиотеки существует две основные особенности:

- практически все классы библиотеки взаимосвязаны и либо возможна прямая их работа в конвейерном режиме, либо это возможно посредством применения промежуточного класса, который встраивает в цепочку некоторый алгоритм;

- применяя шаблоны, можно варьировать параметры, режимы и другую специфику работы алгоритмов (разумеется, там, где это предусмотрено).

Основные функциональные возможности библиотеки

Далее приводятся примеры работы с основными типами классов библиотеки. Эти примеры показывают практически все важные аспекты работы с **Crypto++**.

Фильтры трансформации

Классы, позволяющие создавать фильтры для объектов, основная функциональность которых не связана с конвейерным применением:

- **HashFilter** – фильтры на основе хэш-алгоритма;
- **PK_EncryptorFilter/PK_DecryptorFilter** – фильтры преобразований для алгоритмов с публичным ключом;
- **SignerFilter** – фильтры для получения подписи данных в алгоритмах ЭЦП;
- **SignatureVerificationFilter** – фильтры для верификации подписи данных в ЭЦП;
- **StreamTransformationFilter** – фильтры для работы с симметричными алгоритмами.

Симметричные блочные алгоритмы

К симметричным алгоритмам относятся алгоритмы групп *AES-кандидаты*, *AES*, *DES*, *DES-наследники* и другие. Для корректной работы этих алгоритмов (как и всех остальных) предоставляется полная инфраструктура. Далее приводится пример использования абстрактного симметричного алгоритма [4]:

```
AutoSeededRandomPool rng (true);
byte key[BLOWFISH::DEFAULT_KEYLENGTH];
byte iv[BLOWFISH::BLOCKSIZE];
std::string plainText = “Only text, nothing more.”;
std::string encryptText, decyptText;

rng.GenerateBlock (key, sizeof (key));
rng.GenerateBlock (iv, sizeof (iv));

CFB_Mode<BLOWFISH>::Encryption Encryptor (key, sizeof (key), iv);
StringSource (plainText, true,
              new StreamTransformationFilter (Encryptor,
                new StringSink (encryptText))
             );

CFB_Mode<BLOWFISH>::Decryption Decryptor(key, sizeof (key), iv);
StringSource (encryptText, true,
              new StreamTransformationFilter (Decryptor,
                new SourceSink (decryptText))
             );
```

В этом примере реализован полный «жизненный цикл» некоторого алгоритма:

- сгенерирован случайный ключ **key** и вектор инициализации **iv** необходимой длины;
- с учётом режима шифрования (*CFB*) задан алгоритм шифрования *Blowfish*;
- произведено шифрование открытого текста;
- с учётом режима шифрования задан алгоритм дешифрования;
- из шифрованного текста восстановлен открытый текст.

Асимметричные алгоритмы

Основная отличительная особенность асимметричных алгоритмов (по сравнению с симметричными) заключается в необходимости пристального контроля за ключами. Во-первых, все ключи, как сгенерированные, так и полученные из сторонних источников, должны проходить процедуру проверки. Для этого у ключей предусмотрен специальный метод:

```
Validate (RandomNumberGenerator &rng, unsigned int level)
```

где **rng** – некоторый генератор случайных чисел, а **level** – числовая константа, которая может принимать следующие значения:

- **0** – тест на безопасность использования с программной точки зрения, использование не приведёт к краху выполнения или появлению исключений;
- **1** – ключ проверяется на корректность вероятностными методами, не проводится проверка на слабые ключи и т.п.;
- **2** – позволяет убедиться, что ключ выполняет свои функции корректно, и сделать некоторые разумные тесты на безопасность;
- **3** – максимальная степень проверки.

Во-вторых, при сохранении и загрузке ключей из файлов требуется быть предельно аккуратным, поскольку существует несколько возможных вариантов хранения ключей, несовместимых друг с другом.

Иллюстрация использования асимметричного алгоритма:

```
AutoSeededRandomPool rng;
InvertibleRSAFunction params;
params.GenerateRandomWithKeySize (rng, KEY_LENGTH);
RSA::PrivateKey privateKey (params);
RSA::PublicKey publicKey (params);

if (!privateKey.Validate (rnd, 3) || !publicKey.Validate (rnd, 3))
  throw runtime_error ("Private/public key generation failed");
std::string plainText = “Only text, nothing more.”;
std::string encryptText, decyptText;

RSAES< OAEP<SHA> >::Encryptor Encryptor (publicKey);
StringSource (plainText, true,
              new PK_EncryptorFilter (rng, Encryptor,
                                      new StringSink (encryptText))
             );

RSAES< OAEP<SHA> >::Decryptor Decryptor (privateKey);
StringSource (encryptText, true,
              new PK_DecryptorFilter (rng, Decryptor,
                                      new StringSink (decryptText))
             );
```

При выборе длины ключа необходимо учитывать следующее: чем длиннее ключ, тем больше времени будет затрачено на его генерацию и тем более валидацию. В официальном вики указаны следующие значения пар длина/время генерации для алгоритма *RSA* (см. таблицу 1):

Таблица 1 – Зависимость времени генерации ключа от его длины

Длина ключа, бит	61140	30720	15360	11776	8192	4096	2048	1024	512	256
Время генерации, с	25654.01	2255.30	285.05	142.52	43.08	0.70	0.09	0.01	<0.01	<0.01

Алгоритмы вычисления подписи

Для работы с цифровыми подписями необходим тот же набор действий, что и для асимметричных алгоритмов. Единственное отличие – это нехарактерная для других типов алгоритмов процедура валидации подписи. Не касаясь алгоритмов генерации и валидации ключей, которые неотличимы от аналогичных в асимметричных алгоритмах, следует рассмотреть процедуры подписывания и верификации.

```
RSA::PrivateKey PrivateKey;
RSASS<PSS, SHA1>::Signer signer (PrivateKey);
std::string signature, recovered, message =
    “Only text, nothing more.”;

StringSource (message, true,
              new SignerFilter (rng, signer,
                                new StringSink (signature)
                               )
             );
```

```
RSA::PublicKey PublicKey;
RSASS<PSS, SHA1>::Verifier verifier (PublicKey);
try
{
 StringSource (message + signature, true,
               new SignatureVerificationFilter (verifier,
                                new StringSink (recovered),
                                THROW_EXCEPTION | PUT_MESSAGE)
              );
 cout << “Verification complete successful”;
}
catch (const Exception& e)
{
 cerr << “Error: ” << e.what ();
}
```

Помимо основного назначения (проверки подписи) процедура верификации может быть сопряжена с восстановлением оригинальных данных, для этого необходимо установить объект-приёмник (**`new StringSink`** в приведённом выше примере) и указать соответствующий флаг (**`PUT_MESSAGE`**). В случае же, когда это не требуется, в качестве приёмника необходимо передать **`NULL`**.

Алгоритмы вычисления хэш-значений

Алгоритмы данного класса – самые простые в применении. Всё, что необходимо для того, чтобы включить поддержку этих алгоритмов в свой код, уже было описано выше, поэтому можно ограничиться небольшим примером:

```
SHA256 hash;
std::string digest, message = “Only text, nothing more.”;
StringSource (message, true,
              new HashFilter (hash,
                new HexEncoder (new StringSink (digest))
              )
             );
```

Задание

I. Найти криптографическую библиотеку.
II. Реализовать приложение с графическим интерфейсом, позволяющее выполнять следующие действия.
    1. Шифровать и дешифровать выбранный пользователем файл с использованием одного или нескольких симметричных алгоритмов шифрования:
        1) результаты шифрования и дешифрования должны сохраняться в файлы;
        2) требуемые параметры шифрования, такие как ключ, вектор инициализации и пр., должны считываться из файла.
    2. Шифровать и дешифровать выбранный пользователем файл с использованием одного асимметричного алгоритма шифрования:
        1) реализовать процедуру генерации пары открытый–закрытый ключ;
        2) результаты шифрования и дешифрования должны сохраняться в файлы;
        3) требуемые параметры шифрования должны считываться из файла.
    3. Вычислять и проверять электронную цифровую подпись для выбранного пользователем файла с использованием одного из алгоритмов:
        1) результаты вычисления подписи должны сохраняться в файлы;
        2) требуемые параметры вычисления и проверки подписи должны считываться из файла.
    4. Вычислять хэш-значения для выбранного пользователем файла по одному или нескольким алгоритмам хэширования, при этом вычисленное хэш-значение должно сохраняться в файл.
III. Протестировать правильность работы реализованного приложения.

Дополнительные критерии оценивания качества работы

1. Количество алгоритмов:
   1 – программа суммарно использует не менее 8 алгоритмов, при этом не менее 1 алгоритма для каждого типа криптопреобразований;
   0 – программа использует не менее 1 алгоритма для каждого типа криптопреобразований;
   л.р. не принимается – программа не реализует какой-либо тип криптопреобразований.

Контрольные вопросы

1. Какие действия нужно выполнить при шифровании симметричным алгоритмом?
2. Какие действия нужно выполнить при шифровании асимметричным алгоритмом?
3. Какие действия нужно выполнить при вычислении и проверке электронной цифровой подписи?
4. Какие действия нужно выполнить при хэшировании?

Список литературы

1. Crypto++ Library [Электронный ресурс]. – Режим доступа: http://www.cryptopp.com.
2. Crypto++ Library [Электронный ресурс] : лицензионное соглашение. – Режим доступа: http://www.cryptopp.com/License.txt.
3. Crypto++ Library. Pipelining [Электронный ресурс] : wiki. – Режим доступа: http://www.cryptopp.com/wiki/Pipelining.
4. Lancaster, G. Compiling and Integrating Crypto++ into the Microsoft Visual C++ Environment [Электронный ресурс] / G. Lancaster, J. Walton. – Режим доступа: http://www.codeproject.com/KB/tips/CryptoPPIntegration.aspx.