Отчет по выполнению проекта «Применение методов шифрования для защищённого обмена данными в сети»

Еганян Алексей Б01-904
Зорин Евгений Б01-909
Лозновенко Владислав Б01-903

1. Введение

Целью проекта является реализация протокола PGP для защищенной передачи данных пользователя серверу. Часто приложениям необходимо собирать некоторую информацию у пользователей (легально), например, статистику использования, или логи работы для исправления возникших проблем. Для этого необходимо иметь надежный и быстрый протокол обмена данными. Одним из самых популярных протоколов защищенного обмена данными является PGP - *Pretty Good Privacy*.

2. Pretty Good Privacy

Система PGP (Pretty Good Privacy) - способ передачи файлов по сети, обеспечивающий аутентификацию, конфиденциальность и целостность сообщений.

Каждый пользователь обладает связкой из пар закрытых и открытых ключей, который используются как для расшифрования получаемых сообщений, так и для создания электронных подписей отправляемых сообщений. Пользователь также хранит открытые ключи всех других участников системы для шифрование отправляемых им сообщений.

В системе PGP сообщение подписывается закрытым ключом отправителя, шифруется с использованием симметричного блочного шифрования на случайном ключе. Этот ключ шифруется асимметричной криптосистемой с применением открытого ключа того, кому это сообщение направляется.

Набор закрытых ключей хранится у пользователя, который также можно защитить с использоватнием шифрования. Пользователь выбирает случайный ключ из набора своих закрытых ключей и подписывает с его помощью сообщение (обычно с использованием какой-либо криптографический хэш-функции), объединяет сообщение с подписью и индексом использованного ключа. Затем данные могут опционально пройти сжатие. После этого генерируется сессионный ключ симметричного блочного шифрования, которым шифруется сообщение. На финальном этапе выбирается произвольный открытый ключ из набора адресата,

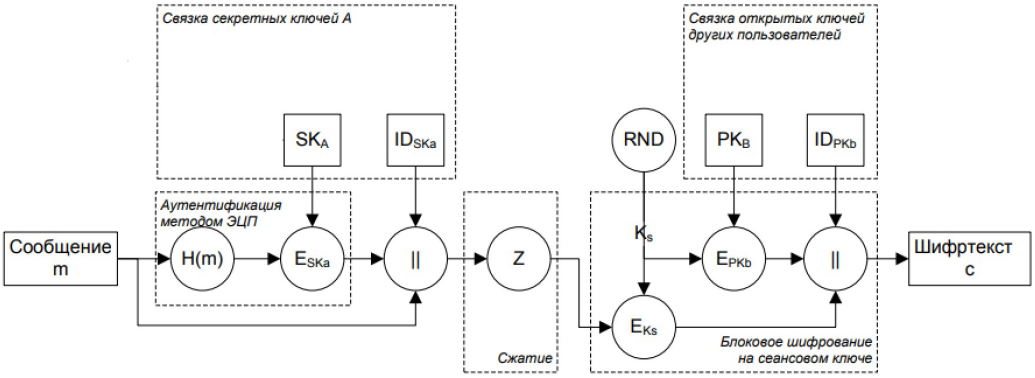

Рис. 1. Схема системы PGP

которым шифруется сессионный ключ. Окончательно в сообщение добавляется зашифрованный сессионный ключ и индекс использованного открытого ключа. Схема шифрования также представлена на рисунке.

- SK_A - закрытый ключ отправителя;
- ID_{SK_A} - идентификатор закрытого ключа, по которому адресат поймёт, какой открытый ключ использовать для проверки подписи;
- m - сообщений для передачи;
- $H(m)$ - криптографическая хэш-функция;
- E_{SK_A} - схема электронной подписи на секретном ключе;
- Z - опциональное сжатие;
- RND - криптографический генератор псевдослучайных чисел;
- K_S - сгенерированный сессионный ключ;
- E_{K_S} - блочное симметричное шифрование на сессионном ключе;
- PK_B - открытый ключ получателя;
- ID_{PK_B} - идентификатор открытого ключа получателя, чтобы он мог понять, который закрытый ключ использовать для расшифрования;
- E_{PK_B} - шифрование сеансового ключа на открытом ключе;
- c - зашифрованное сообщение.

3. Библиотека для шифрования

Для реализации шифрования и электронной подписи была разработана библиотека на C++, которая абстрагирует понятия шифра и ключа. С помощью этой библиотеки можно написать любой алгоритм шифрования и абстрактно работать

с ними. Реализовано это с помощью интерфейсных классов, названия которых говорят сами за себя: *IKey*, *IKeyGenerator*, *IEncryptor*, *IDecryptor*. Интерфейс *IData* необходим для сериализации (т.е. перевода в байты) любого типа, что позволяет библиотеке работать только с массивами байт. Для реализации цифровой подписи существуют классы *ISignatureCreator*, *ISignatureChecker*. Так же подписи часто используются совместно с хэшами, это тоже реализовано с помощью интерфейса *IHash* и классов HashBasedSignatureCreator и *HashBasedSignaturTester*.

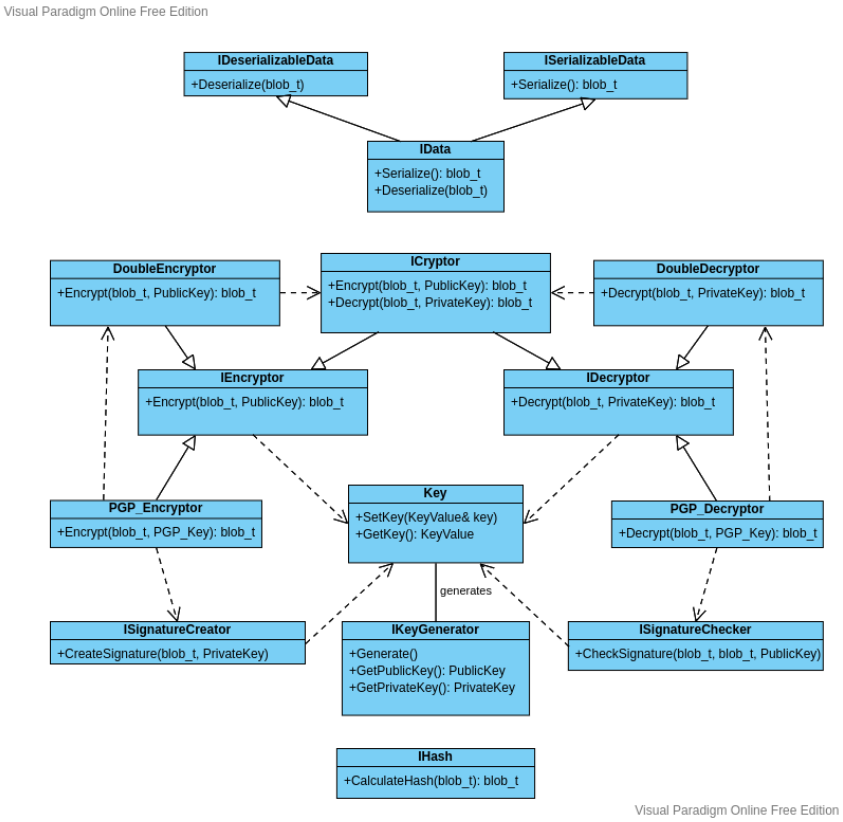

Рис. 2. Диаграмма классов библиотеки

На основе этой библиотеки были реализованы алгоритмы шифрования AES-128, RSA и PGP. Заметим, что интерфейс бибилотеки довольно гибкий, и пользователь может одинаково работать как с шифратором на основе AES-128, так и с более сложным (например, используемое в PGP "двойное"шифрование).

Приведем несколько примеров работы с данной бибилотекой.

- Предположим сначала для простоты, что пользователь имеет массив байт, и ему необходимо его зашифровать, используя AES-128. Это делается очень просто:

```
// создаем генератор ключей
AES128_KeyGenerator KeyGen;
// генерируем ключ
KeyGen.Generate();
// создаем шифратор
AES128_Cryptor AES_128_Cryptor{};
// исходный массив байт
blob_t data = {0x1, 0x2, 0x3, 0x4, 0x5};
// вызываем метод для шифрования
blob_t encryptedData = AES_128_Cryptor.Encrypt(data,
    KeyGen.GetPublicKey());
```

- Пусть теперь необходимо зашифровать исходный файл *input_file.txt* с помощью RSA и записать результат в *output_file.txt*. Для этого реализован специальный интерфейс работы с файлами.

```
// названия файлов
std::string inputFileName = "input_file.txt";
std::string outputFileName = "output_file.txt";
// создаем объект класса для работа с входными файлами
InputFileData inputFileData(inputFileName);

// аналогично генерируем ключи и создаем шифратор RSA
RSA_KeyGenerator KeyGen;
KeyGen.Generate();
RSA_Encryptor RSA_encryptor;

// получаем содержимое файла как массив байт и шифруем их
blob_t data = inputFileData.Serialize();
blob_t encryptedData = RSA_encryptor.Encrypt(data,
    KeyGen.GetPublicKey());

// кладем зашифрованные данные в выходной файл
OutputFileData outputFileData(outputFileName);
outputFileData.Deserialize(blob);
```

- А как расшифровать имеющийся массив данных? Очень просто - вместо шифратора надо использовать дешифратор. Например, посмотрим, как расшифровать то, что лежит в файле *encrypted_ data.txt*

```
// название файла
std::string inputFileName = "encrypted_data.txt";
// создаем объект класса для работа с входными файлами
InputFileData inputFileData(inputFileName);

// аналогично генерируем ключи и создаем дешифратор RSA
RSA_KeyGenerator KeyGen;
KeyGen.Generate();
RSA_Decryptor RSA_decryptor;

// получаем зашифрованный файл как массив байт и расшифровываем его
blob_t encryptedData = inputFileData.Serialize();
blob_t data = RSA_decryptor.Decrypt(encryptedData,
    KeyGen.GetPrivateKey());
```

- Самое интересное здесь то, что работа с PGP осуществляется практически так же. Дополнительно придется указать, какие алгоритмы шифрования и какую хэш функцию мы используем (т.к. PGP не привязан к конкретным алгоритмам), а так же по-другому указать, какие ключи для шифрования использовать (т.к. PGP необходимо несколько ключей для работы, и в общем случае он поддерживает общение с несколькими клиентами). Из-за этих дополнений код может выглядеть весьма громозтко, но это можно исправить с использованием механизмов using. Для упрощения, шаблонные параметры и процесс генерации ключей опустим:

```
// вектор открытых ключей отправителя
std::vector<RSA_Key> sendPrivateKeys;
// вектор закрытых ключей отправителя
std::vector<RSA_Key> sendPublicKeys;

// вектор открытых ключей получателя
std::vector<RSA_Key> resvPrivateKeys;
// вектор закрытых ключей получателя
std::vector<RSA_Key> resvPublicKeys;

AES128_Key sessionKey; // сессионный ключ
/*
Генерация ключей.
```

```
    Сессионный ключ генерируется случайно на стороне отправителя
    */

    PGP_Key encryptKey;
    encryptKey.m_bunchOfPublicKeys = resvPublicKeys;
    encryptKey.m_bunchOfPrivateKeys = sendPrivateKeys;
    encryptKey.m_sessionKey = sessionKey;

    PGP_Encryptor encryptor;
    blob_t encryptedData = encryptor.Encrypt(data, encryptKey);
```

- Так как в примере были опущены шаблоны, могло возникнуть ощущение, что мы реализовали PGP только для использования алгоритма RSA для асимметричного шифрована, AES128 для симметричного, а создание подписи зашили где-то внутри программы. Однако это не так, алгоритмы можно свободно регулировать, используя шаблоны. Так, PGP в качестве шаблонов принимает произвольный алгоритм создания подписи и произвольный "двойной"алгоритм (именуемый у нас как DoubleCryptor) для шифрования. Первый же в свою очередь можно параметризировать алгоритмом расчёта хэш-функции и алгоритмом шифрования её значения, а второй - симметричным алгоритмом шифрования данных и любым алгоритмом шифрования ключа. Мы постарались сделать интерфейс создания PGP шифроваторов максимально удобным для добавления новых реализаций на любых алгоритмах.

4. Производительность

В рамках проекта была измерена эффективность реализованных алгоритмов шифрования AES-128, RSA и PGP, а так же сравнение реализованного протокола PGP с существующей реализацией GnuPG. Графики приведены в конце . Измерялась зависимость времени от размера шифруемых данных.

Проанализируем полученные результаты. AES-128, будучи симметричным шифрованием, показывает ожидаемо хорошие результаты - 30 МБ шифруется примерно за миллисекунду, время шифрования имеет линейную зависимость от размера. Расшифровка происходит на порядок быстрее - 30 МБ за десятые миллисекунды, имеет ту же зависимость (разница в скорости, скорее всего, связаны с особенностями алгоритма и его реализации). RSA, будучи асимметричным шифром, работает, конечно, медленнее - 8 КБ за миллисекунды, так же имеет линейную зависимость; расшифровываем на порядок быстрее из-за разных раз-

меров открытого и закрытого ключей.

Перед анализом PGP посмотрим на DoubleEncrypt, который реализует асимметричное шифрование симметричного ключа и симметричное шифрование данных. Он должен работать примерно столько же, сколько и соответствующий симметричный алгоритм, т.к. занимает большую его часть. В проекте мы использовали DoubleEncryptor для AES-128 и RSA, и время действительно было сопоставимым. Наблюдается один скачок, связанный с особенностями реализации (вызываются методы работы с памятью, которые увеличивают размер выделенной памяти).

Рассмотрим теперь производительность PGP, сравнив с существующей реализацией от GNU - GnuPG. Важное замечание - мы не реализовавыли сжатие данных, которое, по хорошему, должно быть в протоколе PGP, т.к. это могло стать отдельным проектом. Наша реализация показывает неплохие результаты, всего лишь отставая в два раза. Причиной может быть хорошая оптимизация реализации GNU, т.к. мы использовали не самую свежую библиотеку OpenSSL для реализации AES-128 и RSA. Но несмотря на это, для нашей библиотеки это очень хороший результат.

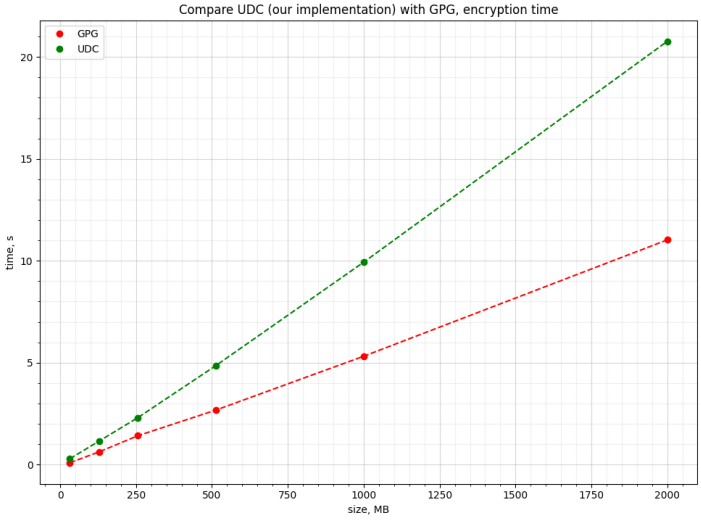

Рис. 3. Сравнение нашей реализации PGP с OpenGP

5. Демонстрация работы программы

Наконец, посмотрим, как данная библиотека может применяться для обмена данных между клиентом и сервером. Для простоты мы сделали сервер, рассылающий широковещательные сообщения в локальной сети с определенным таймаутом, а клиенты при запуске будут получать эти сообщения, подключаться к серверу, обмениваться необходимыми ключами, и далее клиент отправит тестовые данные серверу. В нашем примере эти данные представляют из себя данные о железе клиента, его видеокарте. Это вполне реальный сценарий: приложения часто собирают информацию о том, на каком железе работает программа.

Рис. 4. Демонстрация работы программы, окно клиента

6. Выводы

В рамках проекта мы не только выполнили исходную цель - реализовать защищенную передачу данных между клиентом и сервером - но и попутно сделали полезный инструмент в виде библиотеки C++. Эта библиотека достаточно гибкая и удобная в использовании, а так же позволила реализовать протокол PGP, который показал неплохие результаты в сравнении с существующей реализацией. Проект может быть расширен - как минимум, можно реализовать сжатие данных, а так же подумать над оптимизацией шифрования, например, используя потоки (они не используются в OpenGP).

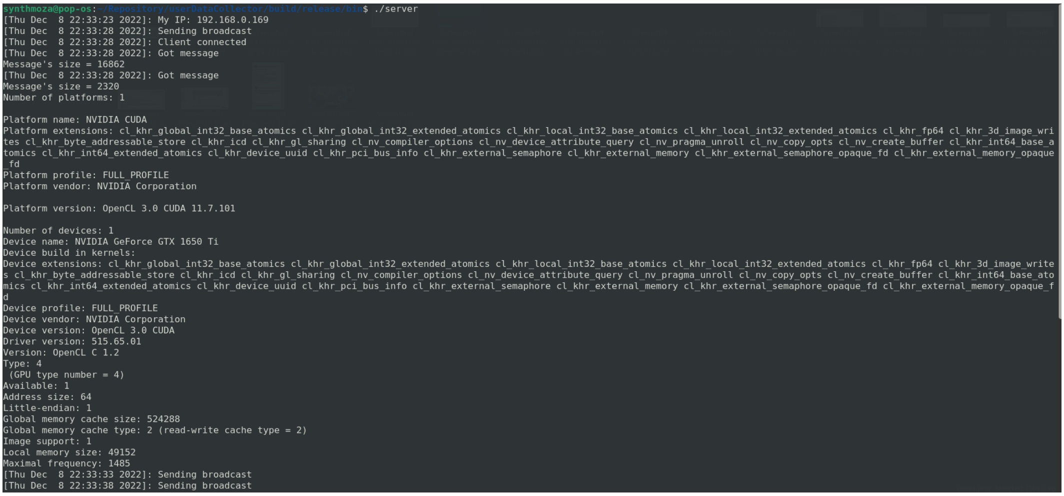

Рис. 5. Демонстрация работы программы, окно сервера

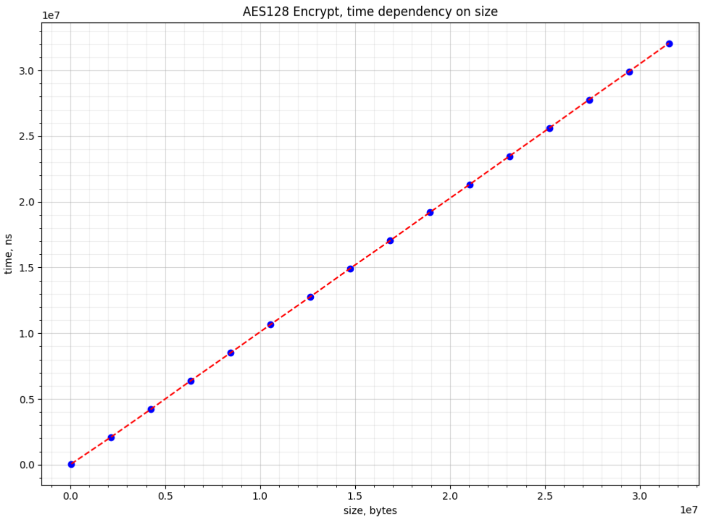

Рис. 6. Исследование производительности шифрования с помощью AES-128

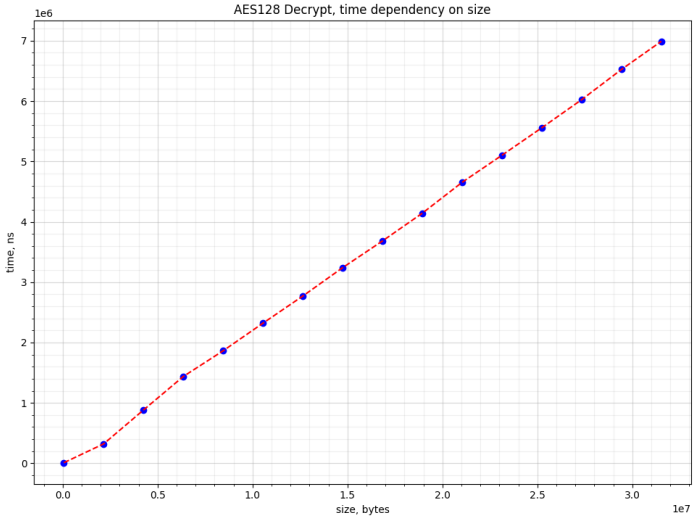

Рис. 7. Исследование производительности расшифрования с помощью AES-128

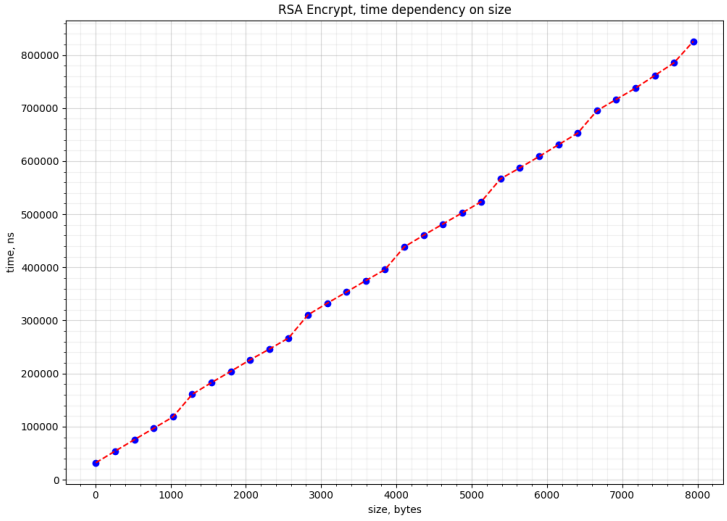

Рис. 8. Исследование производительности шифрования с помощью RSA

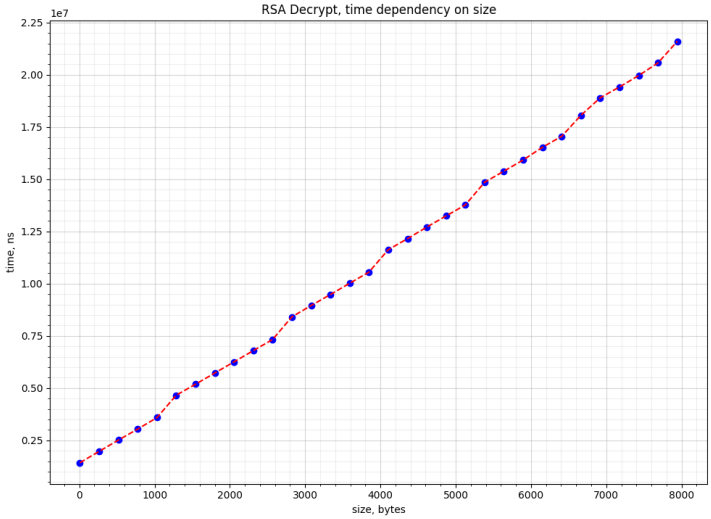

Рис. 9. Исследование производительности расшифрования с помощью RSA

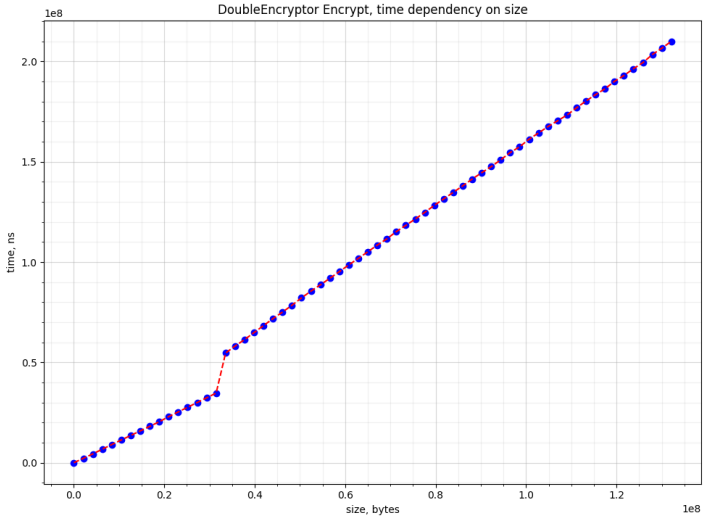

Рис. 10. Исследование производительности шифрования с помощью DoubleEncryptor

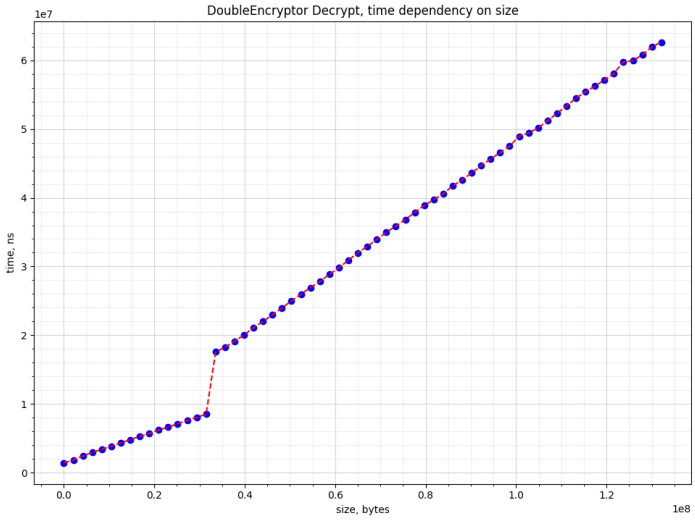

Рис. 11. Исследование производительности расшифрования с помощью DoubleEncryptor