ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

*Кафедра прикладной математики и кибернетики*

Расчётно-графическая работа

Выполнил: студент группы ЗМП-41

Лёвкин И.А.

Проверил:

Фионов А.Н.

Новосибирск, 2025

# Задание

1. Для всех четырёх шифров сравнить длительность функций инициализации, и время генерации ключевого потока.
2. Выбрать одну понравившуюся хеш-функцию и интегрировать ее в программу, как это требуют протоколы выработки секретного ключа. Сравнить время вычисления хеш-функции с временем выполнения других операций в протоколе.
3. Выбрать один понравившийся потоковый шифр и реализовать на его основе приложение для передачи файла в зашифрованном виде по сети с ключом шифрования, полученным по протоколу MQV (или DH, если MQV не был сделан).
4. Провести сравнительный анализ времени выполнения отдельных шагов (фаз) алгоритма (установление ключа, шифрование, передача по сети).

# Ход работы

При выполнении задания было выполнено сравнение функции инициализации и время генерации потоков у шифров НС-128, Rabbit, Salsa20 и Sosemanuk

HS-128

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 bytes | 1000 bytes | 10000 bytes | 100000 bytes | 1000000 bytes |
| Init | 0.000009 | 0.000009 | 0.000009 | 0.000009 | 0.000009 |
| Generation | 0.000000 | 0.000002 | 0.000017 | 0.000172 | 0.001731 |

Rabbit

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 bytes | 1000 bytes | 10000 bytes | 100000 bytes | 1000000 bytes |
| Init | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Generation | 0.000000 | 0.000004 | 0.000035 | 0.000361 | 0.003593 |

Salsa20

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 bytes | 1000 bytes | 10000 bytes | 100000 bytes | 1000000 bytes |
| Init | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Generation | 0.000001 | 0.000005 | 0.000053 | 0.000525 | 0.005242 |

Sosemanuk

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 bytes | 1000 bytes | 10000 bytes | 100000 bytes | 1000000 bytes |
| Init | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Generation | 0.000000 | 0.000002 | 0.000022 | 0.000223 | 0.002237 |

При внедрении SHA-256 в протокол MQV были получены следующие результаты:

|  |
| --- |
| MQV protocol  ------------------------|--------------------  Name | Cycles  ------------------------|--------------------  Alice static keypair | 963628  Bob static keypair | 761736  Alice ephemeral key | 755479  Bob ephemeral key | 754493  Alice MQV shared secret | 77146  Bob MQV shared secret | 64362  Alice's shared secret: 85150cef18daceea5402295d23baf4328db3f7a20ab57dcfd5ab847eaf366327  Bob's shared secret: 85150cef18daceea5402295d23baf4328db3f7a20ab57dcfd5ab847eaf366327  Secrets match: Yes  MQV protocol (SHA-256)  ------------------------|--------------------  Name | Cycles  ------------------------|--------------------  Alice static keypair | 803182  Bob static keypair | 759696  Alice ephemeral key | 756602  Bob ephemeral key | 753406  Alice MQV shared secret | 64906  Bob MQV shared secret | 60350  Alice hashed secret | 1629  Bob hashed secret | 1673  Alice's shared secret: 10d15d426b55d1d426bbe57ac0c55cc3609d0de8ec61b361f64d4dd7c2adeecfe  Bob's shared secret: 10d15d426b55d1d426bbe57ac0c55cc3609d0de8ec61b361f64d4dd7c2adeecfe  Secrets match: Yes  Alice hashed secret: 41049c585b20564ce336ce9b9c2ac03a4c81bf6159c9a1a9e5cff572828a8a36  Bob hashed secret: 41049c585b20564ce336ce9b9c2ac03a4c81bf6159c9a1a9e5cff572828a8a36  Performance  ------------------------|--------------------  Name | Cycles  ------------------------|--------------------  MQV | 3318524  MQV SHA-256 | 3326673  Difference | 8149 |

Как можно видеть разница между протоколом без хеширования и с применением sha-256 не столь значительна – 8149 циклов процессора. Т.к. sha-256 быстрый универсальный алгоритм хеширования, и он занимает малую часть от остальных операций алгоритма ~0.1%.

Далее внедрим потоковый шифр в клиент-серверное приложение для демонстрации.

Протокол: MQV

Хеш функция: SHA-256

Потоковый шифр: Salsa20

**Клиент**

|  |
| --- |
| Connected to server.  Starting MQV key exchange...  Client's shared secret:  db69b85ecfa7de1de17b747d77c4c2eb0e88a10b220c661f5a5e6d26f5fcbf55  Client's shared secret (sha-256):  6cfca6a1b0204bfb96b356b1a92503638f42dba6e2522279e0875abba2e3e1eb  MQV protocol (Client)  ------------------------|--------------------  Name | Cycles  ------------------------|--------------------  Client static key | 17345025  Client ephemeral key | 1071611  Client shared secret | 102953  SHA256 | 8067  Derive key + iv | 71791  Salsa20 init | 220  Encrypt data | 1064314  Send data | 150397514  Encrypted data sent to server(152173 bytes) |

# Сервер

|  |
| --- |
| Server is listening on port 12345  Client connected.  Starting MQV key exchange...  Server's shared secret:  db69b85ecfa7de1de17b747d77c4c2eb0e88a10b220c661f5a5e6d26f5fcbf55  Server's shared secret (sha-256):  6cfca6a1b0204bfb96b356b1a92503638f42dba6e2522279e0875abba2e3e1eb  MQV protocol (Server)  ------------------------|--------------------  Name | Cycles  ------------------------|--------------------  Server static key | 943534  Server ephemeral key | 797130  Server shared secret | 102680  SHA-256 | 18564  Derive key + iv | 95268  Salsa20 init | 306  Receive | 435587668  Decrypt | 1190272  Received (152173 bytes) (decrypted)  saved to received.txt |

Сравнительная таблица

|  |  |  |  |
| --- | --- | --- | --- |
| Компонент Фаза | Сервер | Клиент | Соотношение |
| MQV - Установление ключа |  |  |  |
| Статический ключ | 943,534 | 17,345,025 | 1:18.4 |
| Эфемерный ключ | 797,130 | 1,071,611 | 1:1.34 |
| Общий секрет | 102,680 | 102,953 | ~1:1 |
| Итого MQV | 1,843,344 | 18,519,589 | 1:10 |
| Подготовка шифрования |  |  |  |
| SHA-256 | 18,564 | 8,067 | 2.3:1 |
| Derive key + iv | 95,268 | 71,791 | 1.33:1 |
| Salsa20 init | 306 | 220 | 1.39:1 |
| Итого подготовка | 114,138 | 80,078 | 1.43:1 |
| Обработка данных |  |  |  |
| Шифрование | - | 1,064,314 | - |
| Дешифрование | 1,190,272 | - | - |
| Итого обработка | 1,190,272 | 1,064,314 | 1.12:1 |
| Сетевая передача |  |  |  |
| Передача данных | 435,587,668 | 150,397,514 | 2.9:1 |
| ВСЕГО | 438,735,422 | 170,061,495 | 2.58:1 |

|  |  |  |
| --- | --- | --- |
| Компонент Фаза | Сервер | Клиент |
| MQV - Установление ключа | 0.42% | 10.89% |
| Подготовка шифрования | 0.03% | 0.05% |
| Обработка данных | 0.27% | 0.63% |
| Сетевая передача | 99.28% | 88.43% |

На основе сравнительного анализа времени выполнения алгоритма можно сделать следующие ключевые выводы:

Сетевая передача является основным узким местом системы, занимая более 88% времени выполнения на обеих сторонах. Сервер тратит значительно больше циклов на прием данных (в 2.9 раза), что может быть связано с архитектурными особенностями сетевого стека.

Криптографические операции в целом эффективны, однако наблюдается существенный дисбаланс в производительности между клиентом и сервером.

Клиент тратит в 10 раз больше времени на установление ключа по протоколу MQV, что подтверждает его более слабые вычислительные возможности.

Операции симметричного шифрования (Salsa20) показывают хорошую производительность на обеих сторонах, с минимальным преимуществом сервера в дешифровании.

# Листинг

demo/main.cpp

|  |
| --- |
| #include <rdtsc.h>  #include <chrono>  #include <iomanip>  #include <iostream>  #include <sstream>  #include <string>  #include <vector>  #include "dh.h"  #include "dh\_params.h"  #include "generators.h"  #include "measure.h"  #include "mqv.h"  #include "prime.h"  #include "print.h"  #include "sha256.h"  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void demo\_dh(const std::string& params\_path)  {      mpz\_t alice\_private, alice\_public, bob\_private, bob\_public, alice\_secret,          bob\_secret;      mpz\_inits(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      // Keys      auto alice\_key\_time = measure\_time([&]() {          generate\_private\_key(alice\_private, params.q);          generate\_public\_key(alice\_public, params.g, alice\_private, params.p);      });      auto bob\_key\_time = measure\_time([&]() {          generate\_private\_key(bob\_private, params.q);          generate\_public\_key(bob\_public, params.g, bob\_private, params.p);      });      // Shared secrets      auto alice\_shared\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(alice\_secret, bob\_public, alice\_private, params.p);      });      auto bob\_shared\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(bob\_secret, alice\_public, bob\_private, params.p);      });      print\_performance\_table(          "Diffie-Hellman protocol",          { { "Alice key", alice\_key\_time },              { "Bob key", bob\_key\_time },              { "Alice shared secret", alice\_shared\_secret\_time },              { "Bob shared secret", bob\_shared\_secret\_time } },          NAME\_WIDTH, CYCLES\_WIDTH);      std::cout << std::endl;      print\_secrets(alice\_secret, bob\_secret);      // Cleanup      mpz\_clears(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);  }  void demo\_subgroup\_dh(const std::string& params\_path)  {      mpz\_t alice\_private, alice\_public, bob\_private, bob\_public;      mpz\_t alice\_secret, bob\_secret;      mpz\_inits(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      // Keys      auto alice\_key\_time = measure\_time([&]() {          generate\_private\_key(alice\_private, params.q);          generate\_public\_key(alice\_public, params.g, alice\_private, params.p);      });      auto bob\_key\_time = measure\_time([&]() {          generate\_private\_key(bob\_private, params.q);          generate\_public\_key(bob\_public, params.g, bob\_private, params.p);      });      // Shared secrets      auto alice\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(alice\_secret, bob\_public, alice\_private, params.p);      });      auto bob\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(bob\_secret, alice\_public, bob\_private, params.p);      });      print\_performance\_table(          "Subgroup Diffie-Hellman protocol",          { { "Alice key", alice\_key\_time },              { "Bob key", bob\_key\_time },              { "Alice shared secret", alice\_secret\_time },              { "Bob shared secret", bob\_secret\_time } },          NAME\_WIDTH, CYCLES\_WIDTH);      std::cout << std::endl;      print\_secrets(alice\_secret, bob\_secret);      // Cleanup      mpz\_clears(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);  }  void demo\_mqv(const std::string& params\_path, bool should\_print\_info)  {      mpz\_t alice\_ephemeral\_private, alice\_ephemeral\_public;      mpz\_t bob\_ephemeral\_private, bob\_ephemeral\_public;      mpz\_t alice\_secret, bob\_secret;      mpz\_inits(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      MQVKeyPair alice\_static, bob\_static;      // Static keys      auto alice\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(alice\_static, params); });      auto bob\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(bob\_static, params); });      // Ephemeral keys      auto alice\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(alice\_ephemeral\_private, params.q);          generate\_public\_key(alice\_ephemeral\_public, params.g,              alice\_ephemeral\_private, params.p);      });      auto bob\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(bob\_ephemeral\_private, params.q);          generate\_public\_key(bob\_ephemeral\_public, params.g,              bob\_ephemeral\_private, params.p);      });      // Shared secrets      auto alice\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(alice\_secret, alice\_static,              alice\_ephemeral\_private, alice\_ephemeral\_public,              bob\_ephemeral\_public, bob\_static.public\_key,              params);      });      auto bob\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(bob\_secret, bob\_static,              bob\_ephemeral\_private, bob\_ephemeral\_public,              alice\_ephemeral\_public, alice\_static.public\_key,              params);      });      if (should\_print\_info) {          print\_performance\_table(              "MQV protocol",              { { "Alice static keypair", alice\_static\_time },                  { "Bob static keypair", bob\_static\_time },                  { "Alice ephemeral key", alice\_ephemeral\_time },                  { "Bob ephemeral key", bob\_ephemeral\_time },                  { "Alice MQV shared secret", alice\_secret\_time },                  { "Bob MQV shared secret", bob\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << std::endl;          print\_secrets(alice\_secret, bob\_secret);      }      mpz\_clears(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);  }  void demo\_mqv\_sha256(const std::string& params\_path, bool should\_print\_info)  {      mpz\_t alice\_ephemeral\_private, alice\_ephemeral\_public;      mpz\_t bob\_ephemeral\_private, bob\_ephemeral\_public;      mpz\_t alice\_secret, bob\_secret;      mpz\_inits(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      MQVKeyPair alice\_static, bob\_static;      // Static keys      auto alice\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(alice\_static, params); });      auto bob\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(bob\_static, params); });      // Ephemeral keys      auto alice\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(alice\_ephemeral\_private, params.q);          generate\_public\_key(alice\_ephemeral\_public, params.g,              alice\_ephemeral\_private, params.p);      });      auto bob\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(bob\_ephemeral\_private, params.q);          generate\_public\_key(bob\_ephemeral\_public, params.g,              bob\_ephemeral\_private, params.p);      });      // Shared secrets      auto alice\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(alice\_secret, alice\_static,              alice\_ephemeral\_private, alice\_ephemeral\_public,              bob\_ephemeral\_public, bob\_static.public\_key,              params);      });      auto bob\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(bob\_secret, bob\_static,              bob\_ephemeral\_private, bob\_ephemeral\_public,              alice\_ephemeral\_public, alice\_static.public\_key,              params);      });      SHA256 sha256;      int iterations = should\_print\_info ? 1000 : 1;      // Warmup      // for (int i = 0; i < iterations; i++) {      //     sha256("warmup string");      //     sha256.reset();      // }      std::string alice\_hash = "", bob\_hash = "";      double alice\_hashed\_time = 0, bob\_hashed\_time = 0;      auto alice\_secret\_str = mpz\_get\_str(NULL, 16, alice\_secret);      auto bob\_secret\_str = mpz\_get\_str(NULL, 16, bob\_secret);      for (int i = 0; i < iterations; i++) {          // Hash the shared secrets with SHA-256          alice\_hashed\_time += measure\_time([&]() {              alice\_hash = sha256(alice\_secret\_str);              sha256.reset();          });          bob\_hashed\_time += measure\_time([&]() {              bob\_hash = sha256(bob\_secret\_str);              sha256.reset();          });      }      alice\_hashed\_time /= iterations;      bob\_hashed\_time /= iterations;      if (should\_print\_info) {          print\_performance\_table(              "MQV protocol (SHA-256)",              { { "Alice static keypair", alice\_static\_time },                  { "Bob static keypair", bob\_static\_time },                  { "Alice ephemeral key", alice\_ephemeral\_time },                  { "Bob ephemeral key", bob\_ephemeral\_time },                  { "Alice MQV shared secret", alice\_secret\_time },                  { "Bob MQV shared secret", bob\_secret\_time },                  { "Alice hashed secret", alice\_hashed\_time },                  { "Bob hashed secret", bob\_hashed\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << std::endl;          print\_secrets(alice\_secret, bob\_secret);          std::cout << "Alice hashed secret: " << alice\_hash << "\n";          std::cout << "Bob hashed secret:   " << bob\_hash << "\n";      }      mpz\_clears(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);  }  int main()  {      const std::string params\_dir\_path = "";      const std::string multiplicative\_params\_path          = params\_dir\_path + "multiplicative\_params.txt";      const std::string cyclic\_params\_path = params\_dir\_path + "cyclic\_params.txt";      demo\_mqv(cyclic\_params\_path, true);      demo\_mqv\_sha256(cyclic\_params\_path, true);      double mqv\_time = 0, mqv\_sha256\_time = 0;      int iterations = 500;      for (int i = 0; i < iterations; i++) {          mqv\_time += measure\_time([&]() {              demo\_mqv(cyclic\_params\_path, false);          });          mqv\_sha256\_time += measure\_time([&]() {              demo\_mqv\_sha256(cyclic\_params\_path, false);          });      }      mqv\_time /= iterations;      mqv\_sha256\_time /= iterations;      print\_performance\_table(          "Performance",          {              { "MQV", mqv\_time },              { "MQV SHA-256", mqv\_sha256\_time },              { "Difference", mqv\_sha256\_time - mqv\_time },          },          NAME\_WIDTH, CYCLES\_WIDTH);      return EXIT\_SUCCESS;  } |

**duo\_client**

client.cpp

|  |
| --- |
| #include "client.h"  #include "dh.h"  #include "dh\_params.h"  #include "measure.h"  #include "mqv.h"  #include "network\_session.h"  #include "print.h"  #include "salsa20.h"  #include "sha256.h"  #include <fstream>  #include <helpers.h>  #include <iostream>  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void run\_client(const std::string& params\_path, const std::string& server\_ip, Protocol protocol, const std::string& file\_to\_send)  {      switch (protocol) {      case Protocol::DH:          run\_dh\_client(params\_path, server\_ip);          break;      case Protocol::MQV:          run\_mqv\_client(params\_path, server\_ip);          break;      case Protocol::MQV\_SHA256\_SALSA20:          run\_mqv\_client\_sha256\_salsa20(params\_path, server\_ip, file\_to\_send);          break;      }  }  void run\_dh\_client(const std::string& params\_path, const std::string& server\_ip)  {      NetworkSession session;      session.connect\_to\_server(server\_ip);      std::cout << "Starting Diffie-Hellman key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t client\_private, client\_public, server\_public, client\_secret;      mpz\_inits(client\_private, client\_public, server\_public, client\_secret, NULL);      try {          auto client\_key\_time = measure\_time([&]() {              generate\_private\_key(client\_private, params.q);              generate\_public\_key(client\_public, params.g, client\_private, params.p);          });          if (!session.receive\_value(server\_public)) {              throw std::runtime\_error("Failed to receive server's public key");          }          if (!session.send\_value(client\_public)) {              throw std::runtime\_error("Failed to send public key");          }          auto client\_secret\_time = measure\_time([&]() {              compute\_shared\_secret(client\_secret, server\_public, client\_private, params.p);          });          print\_performance\_table(              "Diffie-Hellman protocol (Client)",              { { "Client key", client\_key\_time },                  { "Client shared secret", client\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nClient's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, client\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(client\_private, client\_public, server\_public, client\_secret, NULL);  }  void run\_mqv\_client(const std::string& params\_path, const std::string& server\_ip)  {      NetworkSession session;      session.connect\_to\_server(server\_ip);      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, server\_ephemeral\_public;      mpz\_t server\_static\_public, client\_secret;      MQVKeyPair client\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);      try {          // Generate static key pair          auto client\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(client\_static\_keypair, params);          });          // Generate ephemeral key pair          auto client\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.receive\_value(server\_static\_public)) {              throw std::runtime\_error("Failed to receive server's static public key");          }          if (!session.send\_value(client\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          // Exchange ephemeral public keys          if (!session.receive\_value(server\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive server's ephemeral public key");          }          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          // Compute shared secret          auto client\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(client\_secret, client\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  server\_ephemeral\_public, server\_static\_public,                  params);          });          print\_performance\_table(              "MQV protocol (Client)",              { { "Client static key", client\_static\_time },                  { "Client ephemeral key", client\_ephemeral\_time },                  { "Client shared secret", client\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nClient's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, client\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);  }  std::vector<uint8\_t> readFile(const std::string& filename)  {      std::ifstream file(filename, std::ios::binary);      return { std::istreambuf\_iterator<char>(file), {} };  }  void run\_mqv\_client\_sha256\_salsa20(const std::string& params\_path, const std::string& server\_ip, const std::string& file\_to\_send)  {      NetworkSession session;      session.connect\_to\_server(server\_ip);      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, server\_ephemeral\_public;      mpz\_t server\_static\_public, client\_secret;      MQVKeyPair client\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);      try {          // Generate static key pair          auto client\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(client\_static\_keypair, params);          });          // Generate ephemeral key pair          auto client\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.receive\_value(server\_static\_public)) {              throw std::runtime\_error("Failed to receive server's static public key");          }          if (!session.send\_value(client\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          // Exchange ephemeral public keys          if (!session.receive\_value(server\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive server's ephemeral public key");          }          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          // Compute shared secret          auto client\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(client\_secret, client\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  server\_ephemeral\_public, server\_static\_public,                  params);          });          SHA256 sha256;          std::string client\_secret\_str = mpz\_get\_str(nullptr, 16, client\_secret);          std::string hashed\_secret = "";          auto sha256\_time = measure\_time([&]() {              hashed\_secret = sha256(client\_secret\_str);          });          uint8\_t key[32];          uint8\_t iv[8];          auto derive\_key\_time = measure\_time([&]() {              derive\_salsa20\_key\_iv(hashed\_secret, key, iv);          });          // Init Salsa20          ECRYPT\_ctx ctx;          auto ecrypt\_init\_time = measure\_time([&]() {              ECRYPT\_init();              ECRYPT\_keysetup(&ctx, key, 256, 64); // keybits=256, ivbits=64 (8 bytes)              ECRYPT\_ivsetup(&ctx, iv);          });          std::cout << "\nClient's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, client\_secret);          std::cout << std::endl;          std::cout << "\nClient's shared secret (sha-256):" << std::endl;          std::cout << hashed\_secret << std::endl;          std::cout << std::endl;          auto plain = readFile(file\_to\_send);          std::vector<uint8\_t> cipher;          cipher.resize(plain.size());          auto encrypt\_time = measure\_time([&]() {              if (!plain.empty()) {                  ECRYPT\_encrypt\_bytes(&ctx, plain.data(), cipher.data(), static\_cast<u32>(plain.size()));              }          });          auto size = cipher.size();          std::vector<uint8\_t> size\_in\_bytes(sizeof(size));          ::memcpy(size\_in\_bytes.data(), &size, sizeof(size));          auto send\_time = measure\_time([&]() {              if (session.send\_data(size\_in\_bytes) < 0) { // Отправляем размер                  throw std::runtime\_error("Failed to send encrypted data size");              }              if (session.send\_data(cipher) < 0) {                  throw std::runtime\_error("Failed to send encrypted data");              }          });          print\_performance\_table(              "MQV protocol (Client)",              { { "Client static key", client\_static\_time },                  { "Client ephemeral key", client\_ephemeral\_time },                  { "Client shared secret", client\_secret\_time },                  { "SHA256", sha256\_time },                  { "Derive key + iv", derive\_key\_time },                  { "Salsa20 init", ecrypt\_init\_time },                  { "Encrypt data", encrypt\_time },                  { "Send data", send\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "Encrypted data sent to server" << "(" << size << " bytes)" << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);  } |

server.cpp

|  |
| --- |
| #include "server.h"  #include "dh.h"  #include "dh\_params.h"  #include "measure.h"  #include "mqv.h"  #include "network\_session.h"  #include "print.h"  #include <fstream>  #include <helpers.h>  #include <iostream>  #include <salsa20.h>  #include <sha256.h>  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void run\_server(const std::string& params\_path, Protocol protocol)  {      switch (protocol) {      case Protocol::DH:          run\_dh\_server(params\_path);          break;      case Protocol::MQV:          run\_mqv\_server(params\_path);          break;      case Protocol::MQV\_SHA256\_SALSA20:          run\_mqv\_server\_sha256\_salsa20(params\_path);          break;      }  }  void run\_dh\_server(const std::string& params\_path)  {      NetworkSession session;      session.start\_server();      std::cout << "Starting Diffie-Hellman key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t server\_private, server\_public, client\_public, server\_secret;      mpz\_inits(server\_private, server\_public, client\_public, server\_secret, NULL);      try {          auto server\_key\_time = measure\_time([&]() {              generate\_private\_key(server\_private, params.q);              generate\_public\_key(server\_public, params.g, server\_private, params.p);          });          if (!session.send\_value(server\_public)) {              throw std::runtime\_error("Failed to send public key");          }          if (!session.receive\_value(client\_public)) {              throw std::runtime\_error("Failed to receive client's public key");          }          auto server\_secret\_time = measure\_time([&]() {              compute\_shared\_secret(server\_secret, client\_public, server\_private, params.p);          });          print\_performance\_table(              "Diffie-Hellman protocol (Server)",              { { "Server key", server\_key\_time },                  { "Server shared secret", server\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nServer's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, server\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(server\_private, server\_public, client\_public, server\_secret, NULL);  }  void run\_mqv\_server(const std::string& params\_path)  {      NetworkSession session;      session.start\_server();      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, client\_ephemeral\_public;      mpz\_t client\_static\_public, server\_secret;      MQVKeyPair server\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);      try {          // Generate static key pair          auto server\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(server\_static\_keypair, params);          });          // Generate ephemeral key pair          auto server\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.send\_value(server\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          if (!session.receive\_value(client\_static\_public)) {              throw std::runtime\_error("Failed to receive client's static public key");          }          // Exchange ephemeral public keys          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          if (!session.receive\_value(client\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive client's ephemeral public key");          }          // Compute shared secret          auto server\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(server\_secret, server\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  client\_ephemeral\_public, client\_static\_public,                  params);          });          print\_performance\_table(              "MQV protocol (Server)",              { { "Server static key", server\_static\_time },                  { "Server ephemeral key", server\_ephemeral\_time },                  { "Server shared secret", server\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nServer's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, server\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);  }  void writeFile(const std::string& filename, const std::vector<uint8\_t>& data)  {      std::ofstream file(filename, std::ios::binary);      file.write(reinterpret\_cast<const char\*>(data.data()), data.size());  }  void run\_mqv\_server\_sha256\_salsa20(const std::string& params\_path)  {      NetworkSession session;      session.start\_server();      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, client\_ephemeral\_public;      mpz\_t client\_static\_public, server\_secret;      MQVKeyPair server\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);      try {          // Generate static key pair          auto server\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(server\_static\_keypair, params);          });          // Generate ephemeral key pair          auto server\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.send\_value(server\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          if (!session.receive\_value(client\_static\_public)) {              throw std::runtime\_error("Failed to receive client's static public key");          }          // Exchange ephemeral public keys          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          if (!session.receive\_value(client\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive client's ephemeral public key");          }          // Compute shared secret          auto server\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(server\_secret, server\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  client\_ephemeral\_public, client\_static\_public,                  params);          });          // Derive key + iv          SHA256 sha256;          std::string server\_secret\_str = mpz\_get\_str(nullptr, 16, server\_secret);          std::string hashed\_secret = "";          auto sha256\_time = measure\_time([&]() {              hashed\_secret = sha256(server\_secret\_str);          });          uint8\_t key[32];          uint8\_t iv[8];          auto derive\_key\_time = measure\_time([&]() {              derive\_salsa20\_key\_iv(hashed\_secret, key, iv);          });          // Init Salsa20          ECRYPT\_ctx ctx;          auto ecrypt\_init\_time = measure\_time([&]() {              ECRYPT\_init();              ECRYPT\_keysetup(&ctx, key, 256, 64); // keybits=256, ivbits=64 (8 bytes)              ECRYPT\_ivsetup(&ctx, iv);          });          std::cout << "\nServer's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, server\_secret);          std::cout << std::endl;          std::cout << "\nServer's shared secret (sha-256):" << std::endl;          std::cout << hashed\_secret << std::endl;          std::cout << std::endl;          size\_t size = 0;          std::vector<uint8\_t> size\_in\_bytes(sizeof(size));          std::vector<uint8\_t> cipher;          auto receive\_time = measure\_time([&]() {              if (!session.receive\_data(size\_in\_bytes, sizeof(size))) {                  throw std::runtime\_error("Failed to receive encrypted data");              }              ::memcpy(&size, size\_in\_bytes.data(), sizeof(size));              if (!session.receive\_data(cipher, size)) {                  throw std::runtime\_error("Failed to receive encrypted data");              }          });          // Дешифровка          std::vector<uint8\_t> plain;          plain.resize(cipher.size());          auto decrypt\_time = measure\_time([&]() {              if (!cipher.empty()) {                  ECRYPT\_decrypt\_bytes(&ctx, cipher.data(), plain.data(), static\_cast<u32>(cipher.size()));              }          });          writeFile("received.txt", plain);          print\_performance\_table(              "MQV protocol (Server)",              {                  { "Server static key", server\_static\_time },                  { "Server ephemeral key", server\_ephemeral\_time },                  { "Server shared secret", server\_secret\_time },                  { "SHA-256", sha256\_time },                  { "Derive key + iv", derive\_key\_time },                  { "Salsa20 init", ecrypt\_init\_time },                  { "Receive", receive\_time },                  { "Decrypt", decrypt\_time },              },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "Received " << "(" << plain.size() << " bytes)" << " (decrypted)" << std::endl                    << "saved to received.txt"                    << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);  } |

helpers.h

|  |
| --- |
| #include "sha256.h"  #include <cstdint>  #include <sstream>  #include <string>  #include <vector>  // hex -> байты  static std::vector<uint8\_t> hex\_to\_bytes(const std::string& hex)  {      std::vector<uint8\_t> out;      out.reserve(hex.size() / 2);      for (size\_t i = 0; i + 1 < hex.size(); i += 2) {          unsigned int byte = 0;          std::istringstream iss(hex.substr(i, 2));          iss >> std::hex >> byte;          out.push\_back(static\_cast<uint8\_t>(byte));      }      return out;  }  // Деривация key (32 байта) и iv (8 байт) из hex-строки SHA256  static void derive\_salsa20\_key\_iv(const std::string& hashed\_secret\_hex, uint8\_t key\_out[32], uint8\_t iv\_out[8])  {      auto hash\_bytes = hex\_to\_bytes(hashed\_secret\_hex);      if (hash\_bytes.size() < 32)          hash\_bytes.resize(32, 0);      for (size\_t i = 0; i < 32; ++i)          key\_out[i] = hash\_bytes[i];      // iv = первые 8 байт SHA256(hashed\_secret\_hex + "IV")      SHA256 sha;      std::string iv\_source = hashed\_secret\_hex + "IV";      std::string iv\_hash\_hex = sha(iv\_source);      auto iv\_hash\_bytes = hex\_to\_bytes(iv\_hash\_hex);      for (size\_t i = 0; i < 8; ++i)          iv\_out[i] = iv\_hash\_bytes[i];  } |