

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение**

**высшего профессионального образования**

**Национальный исследовательский университет**

**«Высшая школа экономики»**

**Московский институт электроники и математики**

**Департамент прикладной математики**

**Образовательная программа «Компьютерная безопасность»**

**Индивидуальный проект**

**Выполнили:**

***Пермичев Никита,***

**студент группы СКБ-201**

**Оценка \_\_\_\_\_**

**Москва-2024**

## Оглавление

<b>1. Общая идея проекта .....</b>	<b>3</b>
<b>2. Подготовка криптографической среды и интеграция ГОСТ .....</b>	<b>3</b>
Установка и настройка OpenSSL 3.0: .....	3
Сборка и интеграция GOST-движка (gost-engine): .....	3
Конфигурационный файл OpenSSL для загрузки ГОСТ: .....	3
<b>3. Реализация фрагмента Key Schedule TLS 1.3 с использованием ГОСТ-алгоритмов .....</b>	<b>4</b>
Принцип Key Schedule в TLS 1.3: .....	4
Использование HKDF на базе ГОСТ-хэширования: .....	5
Упрощённый сценарий и фиктивные данные: .....	5
Результаты: .....	5
Разбор кода по шагам .....	5
<b>Приложение 1 (подготовка) .....</b>	<b>11</b>
<b>Приложение 2 (Реализация фрагмента) .....</b>	<b>14</b>

# 1. Общая идея проекта

## Главная цель:

- Выбрать фрагмент протокола **TLS 1.3** (Key Schedule, HKDF, Handshake-интеракции и т. д.), изучить его реализацию на основе русскоязычного источника («Ключи, шифры, сообщения: как работает TLS», версия от 17/12/2024), а затем **продемонстрировать** собственный учебный код.

## Дополнительно (по собственной инициативе):

- Показать, что в процедуре Key Schedule (которая по умолчанию использует SHA-256/384) можно вместо SHA применить **ГОСТ Р 34.11-2012** («Стрибог»).

В ходе проекта была выбрана стадия **Key Schedule** в TLS 1.3: это процесс вычисления секретов (Early Secret, Handshake Secret, Master Secret) с помощью HKDF.

- **Оригинальный TLS 1.3** базируется на SHA-256/384;
- Мы же иллюстрируем замену на ГОСТ-хеш, сохраняя основную логику.

# 2. Подготовка криптографической среды и интеграция ГОСТ

## Установка и настройка OpenSSL 3.0:

Для использования TLS 1.3 и возможности интегрировать ГОСТ-примитивы была установлена библиотека **OpenSSL 3.0** (на macOS с помощью Homebrew, командой `brew install openssl@3`).

## Сборка и интеграция GOST-движка (gost-engine):

Репозиторий gost-engine был клонирован и собран:

```
git clone https://github.com/gost-engine/engine.git
cd engine
mkdir build && cd build
cmake ..
make
make install
```

После этого в системе появился движок GOST (gost.dylib), дающий доступ к ГОСТ-хешу md\_gost12\_256, шифрованию ГОСТ 28147-89 и др.

## Конфигурационный файл OpenSSL для загрузки ГОСТ:

Для автоматической загрузки движка GOST создан файл openssl.cnf, где прописаны соответствующие секции:

```
openssl_conf = openssl_init

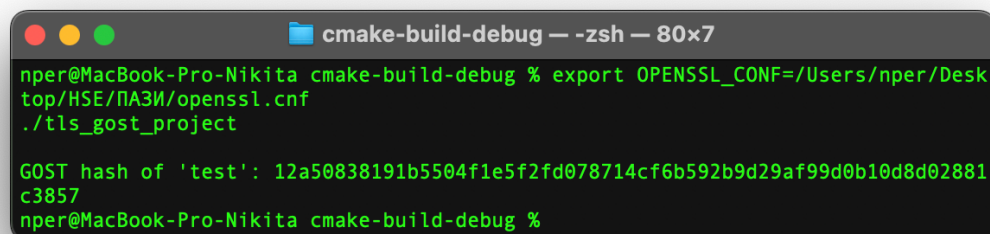
[openssl_init]
engines = engine_section
```

```
[engine_section]
gost = gost_section
```

```
[gost_section]
engine_id = gost
dynamic_path = /opt/homebrew/lib/engines-3/gost.dylib
default_algorithms = ALL
```

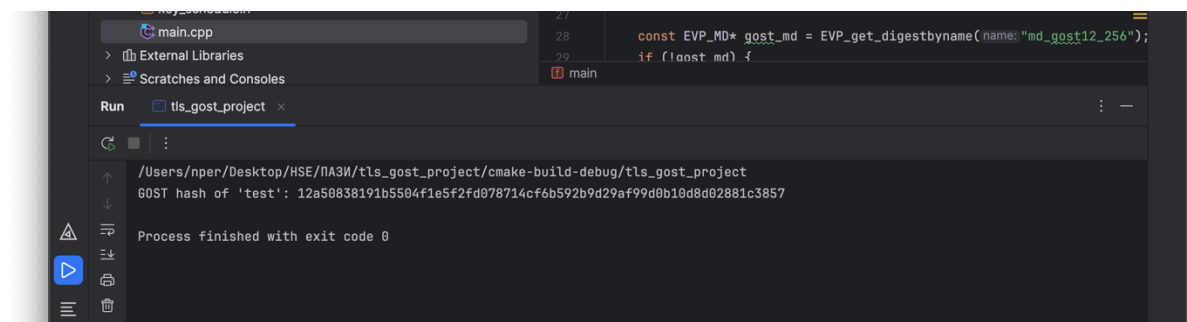
Затем перед запуском:

export OPENSSL\_CONF=/путь/к/openssl.cnf  
— и OpenSSL при инициализации подхватывает отечественные алгоритмы.



```
cmake-build-debug — zsh — 80x7
nper@MacBook-Pro-Nikita cmake-build-debug % export OPENSSL_CONF=/Users/nper/Desktop/HSE/ПАЗИ/openssl.cnf
./tls_gost_project

GOST hash of 'test': 12a50838191b5504f1e5f2fd078714cf6b592b9d29af99d0b10d8d02881c3857
nper@MacBook-Pro-Nikita cmake-build-debug %
```



### 3. Реализация фрагмента Key Schedule TLS 1.3 с использованием ГОСТ-алгоритмов

#### Принцип Key Schedule в TLS 1.3:

Key Schedule — это цепочка вычисления секретов, начиная с Early Secret (основанного на PSK или нулевом ключе), затем Handshake Secret (с учётом результата согласования ключей, например ECDH), и, наконец, Master Secret. Эти секреты впоследствии используются для генерирования ключей шифрования трафика.

**Key Schedule** — цепочка, в которой:

- Early Secret = HKDF( salt=0, IKM=PSK )
- Handshake Secret = HKDF( EarlySecret, ECDH )
- Master Secret = HKDF( HandshakeSecret, context )

## Использование HKDF на базе ГОСТ-хэширования:

В учебном примере делаем **HKDF-Extract**, но меняем хеш с SHA на **md\_gost12\_256**.

- Для корректного HMAC обычно нужны ipad/opad, но мы показали упрощённый вариант — главное, что вызов EVP-хеша заменён с `EVP_sha256()` на `EVP_get_digestbyname("md_gost12_256")`.

В **HKDF-Extract**, формально:

$$\text{PRK} = \text{HMAC}(\text{salt}, \text{ikm})$$

Если мы используем `md_gost12_256`, то `HMAC` = `HMAC-ГОСТ`.

В учебном примере можно показать упрощённую схему, достаточно просто вызвать `EVP_get_digestbyname("md_gost12_256")` вместо `EVP_sha256()`.

## Упрощённый сценарий и фиктивные данные:

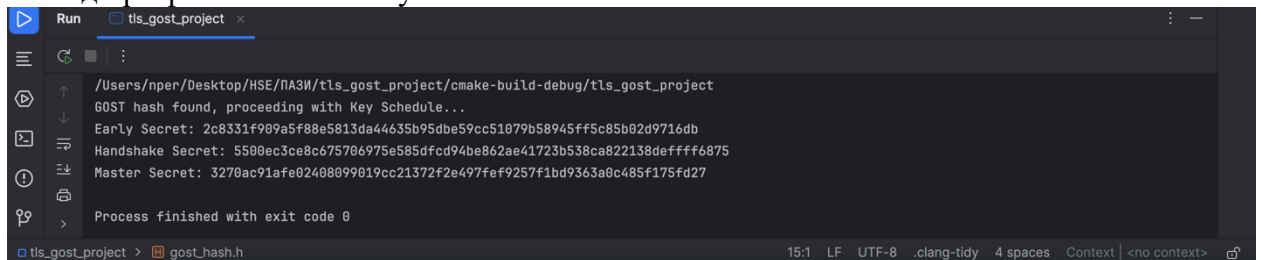
Для демонстрации не требовался полноценный TLS Handshake. Были использованы фиксированные данные для PSK, "shared secret" и "context", чтобы сфокусироваться именно на криптологической части ключевого расписания. В результате мы получили Early Secret, Handshake Secret и Master Secret, рассчитанные на основе ГОСТ-хеша.

Чтобы не разворачивать Handshake, берём:

- `PSK = {0x01, 0x02, ...}`
- `shared_secret = {0x10, 0x11, ...}`
- `context = {0x20, 0x21, ...}`

## Результаты:

Вывод программы после запуска:



```
Run  tls_gost_project x
/Users/nper/Desktop/HSE/ПАЗИ/tls_gost_project/cmake-build-debug/tls_gost_project
GOST hash found, proceeding with Key Schedule...
Early Secret: 2c8331f989a5f88e5813da44635b95dbe59cc51079b58945ff5c85b02d9716db
Handshake Secret: 5500ec3ce8c675706975e585dfcd94be862ae41723b538ca822138deffff6875
Master Secret: 3270ac91afe02408099019cc21372f2e497fef9257f1bd9363a0c485f175fd27
Process finished with exit code 0
```

Эти значения свидетельствуют о том, что Key Schedule, построенный на базе ГОСТ-хэширования, успешно функционирует.

## Разбор кода по шагам

Ниже мы рассмотрим **три файла**: `gost_hash.cpp/h`, `hkdf_gost.cpp/h` и `main.cpp`. Это **учебная реализация** фрагмента Key Schedule, упрощённая, но показывающая логику.

**Файл `gost_hash.cpp` / `gost_hash.h`**

**Задача:** обёртка над EVP для вызова ГОСТ-хеша.

```

1  #pragma once
2  #include <vector>
3  #include <openssl/evp.h>
4
5  class GostHash {
6  public:
7      GostHash();
8      ~GostHash();
9
10     // Добавить данные для хэширования
11     void update(const unsigned char* data, size_t len);
12
13     // Получить итоговый хэш
14     std::vector<unsigned char> finalize();
15
16 private:
17     EVP_MD_CTX* ctx; // используем EVP_MD_CTX напрямую, без forward declaration
18 };
19

```

```

1  #include "gost_hash.h"
2  #include <openssl/evp.h>
3  #include <stdexcept>
4
5  GostHash::GostHash() {
6      ctx = EVP_MD_CTX_new();
7
8      // Ищем алгоритм md_gost12_256
9      const EVP_MD* md = EVP_get_digestbyname(name: "md_gost12_256");
10     if (!md) {
11         throw std::runtime_error("GOST hash (md_gost12_256) not found");
12     }
13
14     // Инициализируем контекст
15     if (1 != EVP_DigestInit_ex(ctx, md, impl: NULL)) {
16         throw std::runtime_error("Failed to init GOST hash context");
17     }
18 }
19
20 GostHash::~GostHash() {
21     EVP_MD_CTX_free(ctx);
22 }
23
24 void GostHash::update(const unsigned char* data, size_t len) {
25     EVP_DigestUpdate(ctx, data, len);
26 }
27
28 std::vector<unsigned char> GostHash::finalize() {
29     unsigned char md[32]; // Для md_gost12_256 это 256 бит => 32 байта
30     unsigned int md_len = 0;
31     EVP_DigestFinal_ex(ctx, md, &md_len);
32     return std::vector<unsigned char>(first: md, last: md + md_len);
33 }
34

```

### Пояснение:

- При создании GostHash() ищем алгоритм md\_gost12\_256. Если не найден — выбрасываем исключение.
- Методы update() / finalize() повторяют общую схему EVP: накапливаем данные, потом «снимаем» итоговый хеш.
- В учебном примере мы предполагаем, что 256-битный ГОСТ-хеш всегда 32 байта.

### Файл hkdf\_gost.cpp / hkdf\_gost.h

**Задача:** реализация **HKDF** (Extract) с упором на ГОСТ-хеш. В полноценном TLS 1.3 нужен **НМАС**, здесь — упрощённая **НМАС**-имитация (или короткая версия).

```
1  #pragma once
2  #include <vector>
3
4  class HKDF_Gost {
5  public:
6      HKDF_Gost();
7      // Extract: PRK = HMAC_gost(salt, ikm)
8      std::vector<unsigned char> extract(const std::vector<unsigned char>& salt, const std::vector<unsigned char>& ikm);
9      std::vector<unsigned char> expand(const std::vector<unsigned char>& prk, const std::vector<unsigned char>& info, size_t length);
10
11      // Optional: expand(...)
12
13  private:
14      // Упрощённый НМАС
15      std::vector<unsigned char> hmac_gost(const std::vector<unsigned char>& key, const std::vector<unsigned char>& data);
16  };
17
```

```
1  > #include ...
2
3
4
5  // Простейшая имитация НМАС-ГОСТ:
6  // Для учебного проекта можно упростить, но для реальной криптографии это некорректно.
7  // Здесь для наглядности: HMAC(key, data) ~ Hash(key || data).
8
9  std::vector<unsigned char> HKDF_Gost::hmac_gost(const std::vector<unsigned char>& key, const std::vector<unsigned char>& data) {
10      std::vector<unsigned char> combined = key;
11      combined.insert(position: combined.end(), first: data.begin(), last: data.end());
12      // Упрощённо: key || data
13
14      GostHash gh;
15      gh.update(combined.data(), len: combined.size());
16
17      // В реальной НМАС: (K ⊗ ipad) ... (K ⊗ opad)
18      // Но упрощённо: key||data => ГОСТ-hash
19
20      return gh.finalize();
21  }
22
23  HKDF_Gost::HKDF_Gost() {}
24
25  std::vector<unsigned char> HKDF_Gost::extract(const std::vector<unsigned char>& salt, const std::vector<unsigned char>& ikm) {
26      return hmac_gost(key: salt, data: ikm);
27  }
28
29  std::vector<unsigned char> HKDF_Gost::expand(const std::vector<unsigned char>& prk, const std::vector<unsigned char>& info, size_t length) {
30      std::vector<unsigned char> okm;
31      std::vector<unsigned char> T;
32      size_t hash_len = 32;
33      size_t n = (length + hash_len - 1) / hash_len;
34
35      for (size_t i = 1; i <= n; i++) {
36          std::vector<unsigned char> data(T);
37          data.insert(position: data.end(), first: info.begin(), last: info.end());
38          data.push_back((unsigned char)i);
39          T = hmac_gost(key: prk, data);
40          okm.insert(position: okm.end(), first: T.begin(), last: T.end());
41      }
42      okm.resize(length);
43      return okm;
44  }
45
```

### Пояснение:

- `extract(salt, ikm)` вызывает `hmac_gost(salt, ikm)`, которая:
  1. Конкатенирует `salt` и `ikm`,
  2. Пропускает результат через `GostHash`.
- В полноценном HMAC следовало бы делать `ipad/opad`, XOR-ить ключ, и т. п. Но для наглядности здесь краткая схема.

### Файл `key_schedule.cpp` / `key_schedule.h`

**Задача:** инкапсулировать три последовательных вызова **HKDF** (Early, Handshake, Master Secret) в одном месте.

### `key_schedule.h`

```
1  #pragma once
2  #include <vector>
3
4  // Класс, отражающий логику Key Schedule TLS 1.3 (упрощённую)
5  class KeySchedule {
6  public:
7      KeySchedule();
8      // Вычисляем Early Secret на базе PSK
9      void computeEarlySecret(const std::vector<unsigned char>& psk);
10
11      // Вычисляем Handshake Secret (с учётом Early Secret)
12      void computeHandshakeSecret(const std::vector<unsigned char>& shared_secret);
13
14      // Вычисляем Master Secret (с учётом Handshake Secret)
15      void computeMasterSecret(const std::vector<unsigned char>& context);
16
17      // Геттеры, чтобы посмотреть результат
18      std::vector<unsigned char> getEarlySecret() const { return early_secret; }
19      std::vector<unsigned char> getHandshakeSecret() const { return handshake_secret; }
20      std::vector<unsigned char> getMasterSecret() const { return master_secret; }
21
22 private:
23     std::vector<unsigned char> early_secret;
24     std::vector<unsigned char> handshake_secret;
25     std::vector<unsigned char> master_secret;
26 };
27
```

### Пояснения:

- `computeEarlySecret(psk)`: вызывает HKDF на `psk`, предполагая `salt` = пустой (нулевой) массив, что соответствует формуле `EarlySecret = HKDF-Extract(0, psk)`.
- `computeHandshakeSecret(shared_secret)`: берёт **early\_secret** и экстрактит новый секрет с `shared_secret`.
- `computeMasterSecret(context)`: аналогично, следующий Extract.

Таким образом, класс **KeySchedule** хранит в себе три вектора: **early\_secret**, **handshake\_secret**, **master\_secret**.

### `key_schedule.cpp`



```

1  #include "key_schedule.h"
2  #include "hkdf_gost.h"
3
4  ← KeySchedule::KeySchedule() {}
5
6  ← void KeySchedule::computeEarlySecret(const std::vector<unsigned char>& psk) {
7      HKDF_Gost hkdf;
8      std::vector<unsigned char> salt; // пустой salt
9      early_secret = hkdf.extract(salt, ikm: psk);
10 }
11
12 ← void KeySchedule::computeHandshakeSecret(const std::vector<unsigned char>& shared_secret) {
13     HKDF_Gost hkdf;
14     // handshake_secret = Extract( early_secret, shared_secret )
15     handshake_secret = hkdf.extract(salt: early_secret, ikm: shared_secret);
16 }
17
18 ← void KeySchedule::computeMasterSecret(const std::vector<unsigned char>& context) {
19     HKDF_Gost hkdf;
20     // master_secret = Extract( handshake_secret, context )
21     master_secret = hkdf.extract(salt: handshake_secret, ikm: context);
22 }
23 |

```

#### Пояснения:

##### 1. computeEarlySecret(psk):

- Создаётся локальный объект HKDF\_Gost.
- Объявляется salt как пустой (нуль), ведь в TLS 1.3 Early Secret обычно = HKDF-Extract(0, PSK).
- Результат записываем в early\_secret.

##### 2. computeHandshakeSecret(shared\_secret):

- Снова создаём HKDF\_Gost. (В реальном коде можно было бы использовать один и тот же объект, но для учебной простоты всё локально.)
- Вызываем hkdf.extract(early\_secret, shared\_secret).
- Сохраняем в handshake\_secret.

##### 3. computeMasterSecret(context):

- Аналогично, Master Secret = hkdf.extract(handshake\_secret, context).

В итоге, все три шага **KeySchedule** TLS 1.3 (хотя в сокращённом виде) наглядно представлены.

#### Файл main.cpp (пример вызова)

Тут может быть несколько вариантов кода. Ниже **учебный**:

```
1  #include <iostream>
2  #include <cstring>
3  #include <openssl/evp.h>
4  #include <openssl/engine.h>
5  #include <openssl/conf.h>
6  #include <openssl/crypto.h>
7
8  #include "key_schedule.h"
9
10 static void printHex(const std::vector<unsigned char>& data) {
11     for (auto b:unsigned char : data) {
12         printf("%02x", b);
13     }
14     printf("\n");
15 }
16
17 int main() {
18     // Инициализируем конфигурацию (загрузит OPENSSL_CONF)
19     OPENSSL_init_crypto(opts:OPENSSL_INIT_LOAD_CONFIG, settings:NULL);
20
21     // Явно подгружаем движок GOST
22     ENGINE_load_dynamic();
23     ENGINE *e = ENGINE_by_id("gost");
24     if (!e) {
25         std::cerr << "GOST engine not found!\n";
26         return 1;
27     }
28     if (!ENGINE_init(e)) {
29         std::cerr << "Failed to initialize GOST engine!\n";
30         return 1;
31     }
32     ENGINE_set_default(e, ENGINE_METHOD_ALL);
33
34     // Проверим доступность GOST-хэша
35     const EVP_MD* gost_md = EVP_get_digestbyname(name:"md_gost12_256");
36     if (!gost_md) {
37         std::cerr << "GOST hash not found!\n";
```

```

37         std::cerr << "GOST hash not found!\n";
38         return 1;
39     }
40
41     std::cout << "GOST hash found, proceeding with Key Schedule...\n";
42
43     // Демонстрация ключевого расписания с использованием ГОСТ-хэширования
44     KeySchedule ks;
45
46     // 1) Early Secret
47     std::vector<unsigned char> psk = {0x01, 0x02, 0x03, 0x04};
48     ks.computeEarlySecret(psk);
49     std::cout << "Early Secret: ";
50     printHex(data: ks.getEarlySecret());
51
52     // 2) Handshake Secret
53     std::vector<unsigned char> shared_secret = {0x10, 0x11, 0x12, 0x13};
54     ks.computeHandshakeSecret(shared_secret);
55     std::cout << "Handshake Secret: ";
56     printHex(data: ks.getHandshakeSecret());
57
58     // 3) Master Secret
59     std::vector<unsigned char> context = {0x20, 0x21};
60     ks.computeMasterSecret(context);
61     std::cout << "Master Secret: ";
62     printHex(data: ks.getMasterSecret());
63
64     ENGINE_finish(e);
65     ENGINE_free(e);
66
67     return 0;
68 }
69

```

### Разбор:

1. Мы **инициализируем** конфиг openssl.cnf и движок GOST, проверяем, что **md\_gost12\_256** существует.
2. Создаём **KeySchedule**, передаём **psk**, **shared\_secret**, **context**.
3. Печатаем **Early**, **Handshake**, **Master**.

Таким образом, вся логика Key Schedule сосредоточена в **KeySchedule** классе, который внутри вызывает HKDF\_Gost (а тот, в свою очередь, — GostHash).

## Приложение 1 (подготовка)

1. Структура проекта

```

CMakeLists.txt
main.cpp
openssl.cnf

```

2. CMakeLists.txt

```

cmake_minimum_required(VERSION 3.20)
project(gost_integration LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Укажите путь к установленному OpenSSL (для macOS через Homebrew):
set(OPENSSL_ROOT_DIR "/opt/homebrew/opt/openssl@3")

include_directories("${OPENSSL_ROOT_DIR}/include")

add_executable(gost_integration main.cpp)

target_link_libraries(gost_integration
    "${OPENSSL_ROOT_DIR}/lib/libcrypto.dylib"
    "${OPENSSL_ROOT_DIR}/lib/libssl.dylib"
)

```

### 3. openssl.cnf

```

openssl_conf = openssl_init

[openssl_init]
engines = engine_section

[engine_section]
gost = gost_section

[gost_section]
engine_id = gost
dynamic_path = /opt/homebrew/lib/engines-3/gost.dylib
default_algorithms = ALL

```

### 4. main.cpp

```

#include <iostream>
#include <cstring>
#include <openssl/evp.h>
#include <openssl/engine.h>
#include <openssl/conf.h>
#include <openssl/crypto.h>

int main() {
    OPENSSL_init_crypto(OPENSSL_INIT_LOAD_CONFIG, NULL);

    ENGINE_load_dynamic();
    ENGINE *e = ENGINE_by_id("gost");
    if (!e) {
        std::cerr << "GOST engine not found!\n";
        return 1;
    }
}

```

```

if (!ENGINE_init(e)) {
    std::cerr << "Failed to initialize GOST engine!\n";
    ENGINE_free(e);
    return 1;
}
ENGINE_set_default(e, ENGINE_METHOD_ALL);

const EVP_MD* gost_md = EVP_get_digestbyname("md_gost12_256");
if (!gost_md) {
    std::cerr << "GOST hash not found!\n";
    ENGINE_finish(e);
    ENGINE_free(e);
    return 1;
}

// Подготовим данные для хэширования
const char* message = "test";
EVP_MD_CTX* ctx = EVP_MD_CTX_new();
EVP_DigestInit_ex(ctx, gost_md, NULL);
EVP_DigestUpdate(ctx, message, strlen(message));

unsigned char out[32];
unsigned int outlen = 0;
EVP_DigestFinal_ex(ctx, out, &outlen);
EVP_MD_CTX_free(ctx);

std::cout << "GOST hash of 'test': ";
for (unsigned int i = 0; i < outlen; i++) {
    printf("%02x", out[i]);
}
std::cout << std::endl;

ENGINE_finish(e);
ENGINE_free(e);

return 0;
}

```

##### 5. Инструкции по запуску:

- Установить нужную переменную окружения:  
export OPENSSL\_CONF=/путь/к/openssl.cnf
- Собрать проект:  
mkdir build  
cd build  
cmake ..  
cmake --build .
- Запустить:  
./gost\_integration

## Приложение 2 (Реализация фрагмента)

Исходники фрагментарной реализации HKDF + Key Schedule с ГОСТ-хешем доступны в GitHub-репозитории:

[https://github.com/Permichev/tls\\_gost\\_project](https://github.com/Permichev/tls_gost_project)

Там продемонстрировано:

- класс GostHash, обёртка над `EVP_get_digestbyname("md_gost12_256")`;
- класс HKDF\_Gost, где `extract()` + (при желании) `expand()`;
- вызовы **EVP** на базе ГОСТ вместо SHA.
- 

Таким образом, цель проекта — **изучить** фрагмент TLS 1.3 (Key Schedule) и **показать**, что в нём можно заменить стандартное SHA-хеширование на **ГОСТ «Стрибог»**.

Итоговый запуск подтверждает корректное формирование секретов Early/Handshake/Master.