Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №5
по дисциплине «Защита информации»

Студент Лукьяненко В.А.

Группа ИУ7-71Б

Преподаватель Руденкова Ю.С.

2025 г.

1 Задание

1.1 Цель работы

Цель работы: разработка и исследование распределённой системы защищённого обмена сообщениями, обеспечивающей криптографическую стойкость, анонимность участников и устойчивость к компрометации, а также анализ уязвимостей реализованных криптографических механизмов и протоколов.

1.2 Содержание работы

Для выполнения данной лабораторной работы необходимо решить следующие задачи:

1. изучить принципы построения защищённых мессенджер-систем и современные требования к криптографической защите информации;
2. разработать архитектуру распределённой системы защищённого обмена сообщениями, включающую клиентов, центр управления ключами и сервер пересылки сообщений;
3. реализовать механизм управления ключами на основе пороговой схемы разделения секрета Шамира с возможностью восстановления ключа из заданного числа долей;
4. реализовать протокол обмена ключами Диффи–Хеллмана на эллиптических кривых с использованием эфемерных ключей и обеспечением совершенной прямой секретности;
5. реализовать механизм анонимной аутентификации клиентов с использованием схемы слепой подписи RSA;
6. реализовать гибридную схему шифрования сообщений с применением асимметричных и симметричных криптографических алгоритмов;

7. обеспечить защиту метаданных и устойчивость к анализу трафика за счёт выравнивания размеров сообщений и генерации фиктивного трафика;
8. реализовать базовые механизмы мониторинга и контроля целостности компонентов системы;
9. провести анализ устойчивости реализованной системы к криптографическим и протокольным атакам;
10. выполнить тестирование разработанной системы и оценить корректность и надёжность её функционирования.

2 Теоретическая часть

Вопросы для защиты работы

1. В чём преимущества пороговой схемы разделения секрета Шамира по сравнению с хранением единого ключа?

Пороговая схема разделения секрета Шамира позволяет разбить секретный ключ на несколько независимых долей таким образом, что восстановление секрета возможно только при наличии заданного минимального количества долей. Основные преимущества данной схемы заключаются в следующем:

— **устойчивость к компрометации** — компрометация одной или нескольких долей не приводит к раскрытию всего секрета;

— **отсутствие единой точки отказа** — отказ или потеря одного узла не нарушает работоспособность системы;

— **гибкость управления доступом** — параметр порога позволяет настраивать уровень отказоустойчивости и безопасности;

— **криптографическая стойкость** — схема основана на интерполяции полинома над конечным полем, что делает восстановление секрета без достаточного числа долей вычислительно невозможным.

Таким образом, пороговая схема Шамира значительно повышает надёжность систем управления ключами по сравнению с централизованным хранением секретов.

2. Почему использование эфемерных ключей в протоколе Диффи–Хеллмана обеспечивает совершенную прямую секретность?

Совершенная прямая секретность (Perfect Forward Secrecy, PFS) означает, что компрометация долгосрочных ключей не приводит к раскрытию ранее переданных сообщений. Использование эфемерных ключей в протоколе Диффи–Хеллмана обеспечивает данное свойство по следующим причинам:

— **уникальность сессионных ключей** — для каждой сессии генерируется новая пара эфемерных ключей;

— **невозможность ретроспективной расшифровки** — даже при компрометации постоянных ключей злоумышленник не сможет восстановить ключи прошлых сессий;

— **кратковременность ключевого материала** — эфемерные ключи уничтожаются после завершения сессии;

— **стойкость к атаке повторного использования ключей** — отсутствует повторяющийся ключевой материал между сессиями.

Благодаря этим свойствам протокол Диффи–Хеллмана с эфемерными ключами широко используется в современных защищённых коммуникационных системах.

3. В чём заключается назначение слепой подписи RSA в системе защищённого обмена сообщениями?

Слепая подпись RSA позволяет подписывать данные без раскрытия их содержимого подписывающей стороне. В контексте защищённой мессенджер-системы данный механизм используется для анонимной аутентификации клиентов. Основные особенности слепой подписи:

— **сохранение анонимности** — центр управления ключами не знает, какой именно открытый ключ он подписывает;

— **подтверждение принадлежности к группе** — наличие подписи подтверждает право клиента участвовать в системе;

— **отделение аутентификации от идентификации** — проверяется факт доверия, но не личность пользователя;

— **устойчивость к подмене ключей** — подпись позволяет проверить, что открытый ключ был авторизован доверенным центром.

Таким образом, слепая подпись RSA обеспечивает баланс между аутентификацией и защитой приватности участников системы.

4. Почему в современных системах используется гибридное шифрование?

Гибридное шифрование сочетает асимметричные и симметричные криптографические алгоритмы, что позволяет объединить их преимущества:

— **эффективность** — симметричные алгоритмы обеспечивают высокую скорость шифрования больших объёмов данных;

— **безопасное распределение ключей** — асимметричные алгоритмы используются для согласования сессионных ключей;

— **масштабируемость** — отсутствие необходимости передачи секретных ключей по защищённому каналу;

— **гибкость алгоритмов** — возможность использовать различные симметричные алгоритмы (AES, ChaCha20) без изменения архитектуры системы.

Гибридный подход является стандартом для большинства современных протоколов защищённой связи.

5. Какие меры позволяют снизить эффективность анализа трафика в защищённых мессенджерах?

Анализ трафика направлен на извлечение метаданных, таких как частота, объём и направление обмена сообщениями. Для снижения его эффективности применяются следующие меры:

— **выравнивание размеров сообщений** — все сообщения приводятся к фиксированной длине;

— **генерация фиктивного трафика** — передача ложных сообщений для маскировки реальной активности;

— **использование временных задержек** — введение случайных пауз между отправками сообщений;

— **шифрование заголовков и служебных данных** — минимизация утечек метаинформации.

Применение данных методов позволяет существенно усложнить построение профиля активности пользователей и повысить уровень их анонимности.

3 Практическая часть.

Листинг 3.1 – Файл common.py,

```python
import os
import json
import base64
import hashlib
from dataclasses import dataclass
from typing import List, Tuple, Optional

from cryptography.hazmat.primitives.asymmetric import ec, rsa
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers.aead import AESGCM,
    ChaCha20Poly1305


# ———————— helpers ————————
def b64e(b: bytes) -> str:
    return base64.b64encode(b).decode("ascii")


def b64d(s: str) -> bytes:
    return base64.b64decode(s.encode("ascii"))


def sha256(b: bytes) -> bytes:
    return hashlib.sha256(b).digest()


def pad_to(data: bytes, size: int) -> bytes:
    if len(data) > size:
        raise ValueError("message too long for fixed padding size")
    return data + b"\x00" * (size - len(data))


def unpad_zeros(data: bytes) -> bytes:
    return data.rstrip(b"\x00")


# ———————— Shamir Secret Sharing over a prime field ————————
```

```python
# A large prime > 2^256; fine for demo. (Not provably best, but works.)
P = (1 << 521) - 1  # Mersenne prime (2^521 - 1)


def _mod_inv(a: int, p: int = P) -> int:
    return pow(a, p - 2, p)


def _eval_poly(coeffs: List[int], x: int, p: int = P) -> int:
    # coeffs: [a0, a1, a2, ...] => a0 + a1*x + a2*x^2 ...
    y = 0
    xp = 1
    for c in coeffs:
        y = (y + c * xp) % p
        xp = (xp * x) % p
    return y


def shamir_split(secret: bytes, n: int = 5, k: int = 3) -> List[Tuple[int, int]]:
    if k < 2 or k > n:
        raise ValueError("bad (n,k)")
    s_int = int.from_bytes(secret, "big")
    if s_int >= P:
        raise ValueError("secret too large for field")

    # random polynomial degree k-1 with constant term = secret
    coeffs = [s_int] + [int.from_bytes(os.urandom(66), "big") % P for _ in range(k - 1)]
    shares = []
    for x in range(1, n + 1):
        y = _eval_poly(coeffs, x)
        shares.append((x, y))
    return shares


def shamir_combine(shares: List[Tuple[int, int]]) -> bytes:
    if len(shares) < 2:
        raise ValueError("need at least 2 shares")
    # Lagrange interpolation at x=0
```

```
76      secret = 0
77      for i, (xi, yi) in enumerate(shares):
78          num = 1
79          den = 1
80          for j, (xj, _yj) in enumerate(shares):
81              if i == j:
82                  continue
83              num = (num * (-xj)) % P
84              den = (den * (xi - xj)) % P
85          li = (num * _mod_inv(den)) % P
86          secret = (secret + yi * li) % P
87
88      # convert back to 32 bytes (master key size in this demo)
89      secret_bytes = secret.to_bytes((secret.bit_length() + 7) // 8,
            "big") or b"\x00"
90      return secret_bytes.rjust(32, b"\x00")[-32:]
91
92
93  # ---------- RSA blind signature (textbook RSA over SHA-256 digest)
        ----------
94  # For lab/demo only. In real systems: use RSA-PSS + proper blind
        signature scheme.
95  @dataclass
96  class RSAPub:
97      n: int
98      e: int
99
100
101 @dataclass
102 class RSAPriv:
103     n: int
104     d: int
105
106
107 def rsa_generate(bits: int = 3072) -> Tuple[RSAPub, RSAPriv]:
108     priv = rsa.generate_private_key(public_exponent=65537,
            key_size=bits)
109     numbers = priv.private_numbers()
110     pubn = numbers.public_numbers.n
111     pube = numbers.public_numbers.e
112     d = numbers.d
```

```
113     return RSAPub(pubn, pube), RSAPriv(pubn, d)
114
115
116 def rsa_blind(msg_hash: bytes, pub: RSAPub) -> Tuple[int, int]:
117     m = int.from_bytes(msg_hash, "big")
118     if m >= pub.n:
119         m = m % pub.n
120     # choose random r coprime to n
121     while True:
122         r = int.from_bytes(os.urandom(64), "big") % pub.n
123         if r > 1 and os.path.exists("/dev/null"):  # no-op to avoid
            lint whining
124             pass
125         if r > 1 and (pow(r, 1, pub.n) != 0) and (hashlib.gcd(r,
            pub.n) == 1):
126             break
127     blinded = (m * pow(r, pub.e, pub.n)) % pub.n
128     return blinded, r
129
130
131 def rsa_sign_int(blinded: int, priv: RSAPriv) -> int:
132     return pow(blinded, priv.d, priv.n)
133
134
135 def rsa_unblind(sig_blinded: int, r: int, pub: RSAPub) -> int:
136     rinv = _mod_inv(r, pub.n) if pub.n != P else pow(r, -1, pub.n)  #
        fallback
137     # correct modular inverse for RSA modulus:
138     rinv = pow(r, -1, pub.n)
139     return (sig_blinded * rinv) % pub.n
140
141
142 def rsa_verify_hash(sig: int, msg_hash: bytes, pub: RSAPub) -> bool:
143     m = int.from_bytes(msg_hash, "big") % pub.n
144     check = pow(sig, pub.e, pub.n)
145     return check == m
146
147
148 # ———————— ECDH P-384 + HKDF to AEAD key ————————
149 def gen_ec_keypair():
150     sk = ec.generate_private_key(ec.SECP384R1())
```

```
151     pk = sk.public_key()
152     pk_bytes = pk.public_bytes(
153         serialization.Encoding.X962,
154         serialization.PublicFormat.UncompressedPoint,
155     )
156     return sk, pk_bytes
157
158
159 def ecdh_derive(sk: ec.EllipticCurvePrivateKey, peer_pk_bytes: bytes,
    context: bytes) -> bytes:
160     peer_pk =
        ec.EllipticCurvePublicKey.from_encoded_point(ec.SECP384R1(),
        peer_pk_bytes)
161     shared = sk.exchange(ec.ECDH(), peer_pk)
162     hkdf = HKDF(
163         algorithm=hashes.SHA384(),
164         length=32,
165         salt=None,
166         info=b"lab2-ecdh-p384|" + context,
167     )
168     return hkdf.derive(shared)  # 32-byte session key
169
170
171 def aead_encrypt(plaintext: bytes, key: bytes, aad: bytes, algo: str =
    "AESGCM") -> Tuple[bytes, bytes]:
172     nonce = os.urandom(12)
173     if algo.upper() == "CHACHA20":
174         aead = ChaCha20Poly1305(key)
175     else:
176         aead = AESGCM(key)
177     ct = aead.encrypt(nonce, plaintext, aad)
178     return nonce, ct
179
180
181 def aead_decrypt(nonce: bytes, ciphertext: bytes, key: bytes, aad:
    bytes, algo: str = "AESGCM") -> bytes:
182     if algo.upper() == "CHACHA20":
183         aead = ChaCha20Poly1305(key)
184     else:
185         aead = AESGCM(key)
186     return aead.decrypt(nonce, ciphertext, aad)
```

```
187
188
189 # ———————— persistence ————————
190 def load_json(path: str, default):
191     if not os.path.exists(path):
192         return default
193     with open(path, "r", encoding="utf-8") as f:
194         return json.load(f)
195
196
197 def save_json(path: str, obj):
198     tmp = path + ".tmp"
199     with open(tmp, "w", encoding="utf-8") as f:
200         json.dump(obj, f, ensure_ascii=False, indent=2)
201     os.replace(tmp, path)
```

Листинг 3.2 – Файл `client.py`,

```
1  import os
2  import json
3  import argparse
4  import asyncio
5  import websockets
6  import math
7
8  from common import (
9      gen_ec_keypair, ecdh_derive,
10     aead_encrypt, aead_decrypt,
11     pad_to, unpad_zeros,
12     sha256, b64e, b64d,
13     load_json, save_json,
14     RSAPub, rsa_verify_hash
15 )
16
17 KMC_URL = "ws://127.0.0.1:8765"
18 MRS_URL = "ws://127.0.0.1:9876"
19 STORE_DIR = "client_store"
20 FIXED_MSG_SIZE = 1024  # padding target
21
22
23 def math_gcd(a, b):
24     return math.gcd(a, b)
```

```
25
26
27 def blind_for_pubkey(pubkey_bytes: bytes, kmc_pub: RSAPub):
28     """
29 ␣␣␣␣Blind␣signature␣over␣H(pubkey_bytes).
30 ␣␣␣␣Return:␣blinded_int_bytes_b64,␣r_int,␣msg_hash
31 ␣␣␣␣"""
32     msg_hash = sha256(pubkey_bytes)
33     m = int.from_bytes(msg_hash, "big") % kmc_pub.n
34     # pick r invertible mod n
35     while True:
36         r = int.from_bytes(os.urandom(64), "big") % kmc_pub.n
37         if r > 1 and math_gcd(r, kmc_pub.n) == 1:
38             break
39     blinded = (m * pow(r, kmc_pub.e, kmc_pub.n)) % kmc_pub.n
40     blinded_b = blinded.to_bytes((blinded.bit_length() + 7) // 8,
       "big")
41     return b64e(blinded_b), r, msg_hash
42
43
44 def unblind(sig_blinded_b64: str, r: int, kmc_pub: RSAPub) -> int:
45     sig_blinded = int.from_bytes(b64d(sig_blinded_b64), "big") %
       kmc_pub.n
46     rinv = pow(r, -1, kmc_pub.n)
47     return (sig_blinded * rinv) % kmc_pub.n
48
49
50 def client_paths(name: str):
51     os.makedirs(STORE_DIR, exist_ok=True)
52     return os.path.join(STORE_DIR, f"{name}.json")
53
54
55 async def kmc_get_pub():
56     async with websockets.connect(KMC_URL, max_size=2_000_000) as ws:
57         await ws.send(json.dumps({"type": "get_pub"}))
58         resp = json.loads(await ws.recv())
59         return RSAPub(n=int(resp["n"]), e=int(resp["e"]))
60
61
62 async def register(name: str):
63     kmc_pub = await kmc_get_pub()
```

```
    sk_long, pk_long = gen_ec_keypair()

    blinded_b64, r, msg_hash = blind_for_pubkey(pk_long, kmc_pub)

    async with websockets.connect(KMC_URL, max_size=2_000_000) as ws:
        await ws.send(json.dumps({"type": "blind_sign", "blinded":
            blinded_b64}))
        resp = json.loads(await ws.recv())
        if resp.get("type") != "blind_sig":
            raise RuntimeError(resp)

    sig = unblind(resp["sig"], r, kmc_pub)

    # verify locally
    if not rsa_verify_hash(sig, msg_hash, kmc_pub):
        raise RuntimeError("KMC signature verification failed
            (client-side)")

    # store private key + pubkey + signature
    st = {
        "name": name,
        "long_sk_pem": sk_long.private_bytes(
            encoding=__import__("cryptography.hazmat.primitives.serializatio
            format=__import__("cryptography.hazmat.primitives.serialization
            encryption_algorithm=__import__("cryptography.hazmat.primitives
        ).decode("utf-8"),
        "long_pk_b64": b64e(pk_long),
        "kmc_sig_int": str(sig),
    }
    save_json(client_paths(name), st)
    print(f"[{name}] registered. long_pk={b64e(pk_long)[:24]}...
        sig={str(sig)[:18]}...")


def load_client(name: str):
    st = load_json(client_paths(name), None)
    if st is None:
        raise RuntimeError(f"no client state for {name}. run
            --register first")
    # load key
    from cryptography.hazmat.primitives import serialization
```

```
    sk = serialization.load_pem_private_key(st["long_sk_pem"].encode("utf-8"),
        password=None)
    return st, sk


async def send_message(sender: str, to: str, text: str, algo: str):
    st, sk_long = load_client(sender)
    kmc_pub = await kmc_get_pub()

    # ephemeral key for PFS
    esk, epk = gen_ec_keypair()

    payload_hello = {
        "kind": "session_init",
        "long_pk_b64": st["long_pk_b64"],
        "kmc_sig_int": st["kmc_sig_int"],
        "eph_pk_b64": b64e(epk),
        "algo": algo,
    }

    async with websockets.connect(MRS_URL, max_size=2_000_000) as ws:
        await ws.send(json.dumps({"type": "hello", "name": sender}))
        _ = await ws.recv()

        # send session init
        await ws.send(json.dumps({"type": "relay", "to": to,
            "payload": payload_hello}))
        _ = await ws.recv()

        # wait for session response
        msg = json.loads(await ws.recv())
        if msg.get("type") != "inbox":
            raise RuntimeError(msg)
        resp = msg["payload"]
        if resp.get("kind") != "session_resp":
            raise RuntimeError(resp)

        peer_eph = b64d(resp["eph_pk_b64"])
        context = (sender + "->" + to).encode("utf-8")
        key = ecdh_derive(esk, peer_eph, context)
```

```
139
140         aad = (sender + "|" + to).encode("utf-8")
141         pt = pad_to(text.encode("utf-8"), FIXED_MSG_SIZE)
142         nonce, ct = aead_encrypt(pt, key, aad=aad, algo=algo)
143
144         await ws.send(json.dumps({"type": "relay", "to": to,
                "payload": {
145              "kind": "msg",
146              "nonce_b64": b64e(nonce),
147              "ct_b64": b64e(ct),
148              "algo": algo,
149              "pad": FIXED_MSG_SIZE
150         }}))
151         _ = await ws.recv()
152         print(f"[{sender}] sent to {to}: {text!r}")
153
154
155 async def listen(name: str):
156     st, _sk_long = load_client(name)
157     kmc_pub = await kmc_get_pub()
158
159     # session cache: peer -> (my_eph_sk, algo)
160     sessions = {}
161
162     async with websockets.connect(MRS_URL, max_size=2_000_000) as ws:
163         await ws.send(json.dumps({"type": "hello", "name": name}))
164         _ = await ws.recv()
165         print(f"[{name}] listening on MRS (Ctrl+C to stop)")
166
167         # dummy traffic: every ~8-20 sec, send a noise packet to
                self (demo)
168         async def dummy():
169             while True:
170                 await asyncio.sleep(8 + int.from_bytes(os.urandom(1),
                        "big") % 13)
171                 try:
172                     await ws.send(json.dumps({"type": "relay", "to":
                            name, "payload": {"kind": "noise"}}))
173                     _ = await ws.recv()
174                 except Exception:
175                     return
```

```
176
177         dummy_task = asyncio.create_task(dummy())
178
179         try:
180             async for raw in ws:
181                 msg = json.loads(raw)
182                 if msg.get("type") != "inbox":
183                     continue
184                 frm = msg["from"]
185                 payload = msg["payload"]
186
187                 if payload.get("kind") == "noise":
188                     continue
189
190                 if payload.get("kind") == "session_init":
191                     long_pk = b64d(payload["long_pk_b64"])
192                     sig_int = int(payload["kmc_sig_int"])
193                     algo = payload.get("algo", "AESGCM")
194
195                     # verify group membership: KMC signature
                        over H(long_pk)
196                     ok = rsa_verify_hash(sig_int, sha256(long_pk),
                        kmc_pub)
197                     if not ok:
198                         print(f"[{name}] REJECT session_init from
                            {frm}: invalid KMC signature")
199                         continue
200
201                     # make my ephemeral for this peer
202                     esk, epk = gen_ec_keypair()
203                     sessions[frm] = (esk, algo)
204
205                     # reply with my ephemeral
206                     await ws.send(json.dumps({"type": "relay", "to":
                        frm, "payload": {
207                         "kind": "session_resp",
208                         "eph_pk_b64": b64e(epk),
209                         "algo": algo
210                     }}))
211                     _ = await ws.recv()
212                     print(f"[{name}] session established with {frm}
```

```
                    (PFS, {algo})")
213                 continue
214
215             if payload.get("kind") == "msg":
216                 if frm not in sessions:
217                     print(f"[{name}] got msg from {frm} but no
                        session")
218                     continue
219                 esk, algo = sessions[frm]
220
221                 # derive key using senders ephemeral? (we
                    dont have it here)
222                 # For demo simplicity: senders eph is only used
                    on sender side,
223                 # receiver uses its own esk + sender eph that came
                    in session_init.
224                 # We stored esk, but need peer eph. So store peer
                    eph at init:
225                 # => minimal hack: peer eph is not persisted. In
                    this demo,
226                 # receiver derives key from its esk and a   fixed
                    context only.
227                 # To keep it correct, well derive key from
                    *last* peer eph saved in sessions as extra data.
228                 # (See below improvement: store peer eph during
                    init.)
229
230                 # -- improvement stored in sessions as dict --
231                 # We'll reconstruct by storing peer eph in
                    sessions2:
232                 pass
233
234         finally:
235             dummy_task.cancel()
236
237
238 # -- Fix listen() session storage properly (store peer eph, then
    decrypt) --
239 async def listen_fixed(name: str):
240     st, _sk_long = load_client(name)
241     kmc_pub = await kmc_get_pub()
```

```
242
243     # peer -> dict
244     sessions = {}
245
246     async with websockets.connect(MRS_URL, max_size=2_000_000) as ws:
247         await ws.send(json.dumps({"type": "hello", "name": name}))
248         _ = await ws.recv()
249         print(f"[{name}] listening on MRS (Ctrl+C to stop)")
250
251         async def dummy():
252             while True:
253                 await asyncio.sleep(8 + int.from_bytes(os.urandom(1),
                        "big") % 13)
254                 try:
255                     await ws.send(json.dumps({"type": "relay", "to":
                            name, "payload": {"kind": "noise"}}))
256                     _ = await ws.recv()
257                 except Exception:
258                     return
259
260         dummy_task = asyncio.create_task(dummy())
261
262         try:
263             async for raw in ws:
264                 msg = json.loads(raw)
265                 if msg.get("type") != "inbox":
266                     continue
267                 frm = msg["from"]
268                 payload = msg["payload"]
269
270                 if payload.get("kind") == "noise":
271                     continue
272
273                 if payload.get("kind") == "session_init":
274                     long_pk = b64d(payload["long_pk_b64"])
275                     sig_int = int(payload["kmc_sig_int"])
276                     algo = payload.get("algo", "AESGCM")
277                     peer_eph = b64d(payload["eph_pk_b64"])
278
279                     ok = rsa_verify_hash(sig_int, sha256(long_pk),
                            kmc_pub)
```

```
280                     if not ok:
281                         print(f"[{name}] REJECT session_init from
                                {frm}: invalid KMC signature")
282                         continue
283
284                     esk, epk = gen_ec_keypair()
285                     sessions[frm] = {"esk": esk, "algo": algo,
                            "peer_eph": peer_eph}
286
287                     await ws.send(json.dumps({"type": "relay", "to":
                            frm, "payload": {
288                         "kind": "session_resp",
289                         "eph_pk_b64": b64e(epk),
290                         "algo": algo
291                     }}))
292                     _ = await ws.recv()
293                     print(f"[{name}] session established with {frm}
                            (PFS, {algo})")
294                     continue
295
296                 if payload.get("kind") == "msg":
297                     sess = sessions.get(frm)
298                     if not sess:
299                         print(f"[{name}] got msg from {frm} but no
                                session")
300                         continue
301
302                     esk = sess["esk"]
303                     algo = payload.get("algo", sess["algo"])
304                     peer_eph = sess["peer_eph"]
305
306                     context = (frm + "->" + name).encode("utf-8")
307                     key = ecdh_derive(esk, peer_eph, context)
308                     aad = (frm + "|" + name).encode("utf-8")
309
310                     nonce = b64d(payload["nonce_b64"])
311                     ct = b64d(payload["ct_b64"])
312                     pt = aead_decrypt(nonce, ct, key, aad=aad,
                            algo=algo)
313                     text = unpad_zeros(pt).decode("utf-8",
                            errors="replace")
```

```
314                     print(f"[{name}] FROM {frm}: {text}")
315                     continue
316
317         finally:
318             dummy_task.cancel()
319
320
321 async def tss_demo():
322     async with websockets.connect(KMC_URL, max_size=2_000_000) as ws:
323         await ws.send(json.dumps({"type": "tss_recover_demo", "which":
                [1, 3, 5]}))
324         print("[TSS demo]", await ws.recv())
325
326
327 def main():
328     ap = argparse.ArgumentParser()
329     ap.add_argument("--name", required=True)
330     ap.add_argument("--register", action="store_true")
331     ap.add_argument("--listen", action="store_true")
332     ap.add_argument("--send", nargs=2, metavar=("TO", "TEXT"))
333     ap.add_argument("--algo", default="AESGCM", choices=["AESGCM",
            "CHACHA20"])
334     ap.add_argument("--tss-demo", action="store_true")
335     args = ap.parse_args()
336
337     async def runner():
338         if args.tss_demo:
339             await tss_demo()
340         if args.register:
341             await register(args.name)
342         if args.listen:
343             await listen_fixed(args.name)
344         if args.send:
345             to, text = args.send
346             await send_message(args.name, to, text, args.algo)
347
348     asyncio.run(runner())
349
350
351 if __name__ == "__main__":
352     main()
```

Листинг 3.3 – Файл kmc.py,

```python
import os
import json
import asyncio
import websockets

from common import (
    rsa_generate, RSAPub, RSAPriv,
    shamir_split, shamir_combine,
    sha256, b64e, b64d,
    load_json, save_json
)

KMC_STATE = "kmc_state.json"
SHARE_DIR = "kmc_shares"
HOST = "127.0.0.1"
PORT = 8765


def init_state():
    st = load_json(KMC_STATE, None)
    if st is None:
        pub, priv = rsa_generate(3072)
        master = os.urandom(32)
        shares = shamir_split(master, n=5, k=3)

        os.makedirs(SHARE_DIR, exist_ok=True)
        for (x, y) in shares:
            save_json(os.path.join(SHARE_DIR, f"share_{x}.json"),
                {"x": x, "y": str(y)})

        st = {
            "rsa_pub": {"n": str(pub.n), "e": pub.e},
            "rsa_priv": {"n": str(priv.n), "d": str(priv.d)},
            "master_sha256": b64e(sha256(master)),  # контроль восстано
                вления TSS
        }
        save_json(KMC_STATE, st)
    return st


```

```
STATE = init_state()
PUB = RSAPub(n=int(STATE["rsa_pub"]["n"]),
    e=int(STATE["rsa_pub"]["e"]))
PRIV = RSAPriv(n=int(STATE["rsa_priv"]["n"]),
    d=int(STATE["rsa_priv"]["d"]))


async def handler(ws):
    """
␣␣␣␣Protocol␣(JSON):
␣␣␣␣-␣{"type":"get_pub"}␣->␣{"type":"pub","n":"...","e":65537}
␣␣␣␣-␣{"type":"blind_sign","blinded":"<b64 int bytes>"}␣->␣
    {"type":"blind_sig","sig":"<b64 int bytes>"}
␣␣␣␣-␣{"type":"tss_recover_demo","which":[1,3,5]}␣->␣
    {"type":"tss_ok","master_sha256":"..."}␣␣(demo)
␣␣␣␣"""
    async for msg in ws:
        try:
            req = json.loads(msg)
            t = req.get("type")

            if t == "get_pub":
                await ws.send(json.dumps({"type": "pub", "n":
                    str(PUB.n), "e": PUB.e}))
                continue

            if t == "blind_sign":
                blinded_b = b64d(req["blinded"])
                blinded_int = int.from_bytes(blinded_b, "big") % PUB.n
                sig_int = pow(blinded_int, PRIV.d, PRIV.n)
                sig_b = sig_int.to_bytes((sig_int.bit_length() + 7) //
                    8, "big")
                await ws.send(json.dumps({"type": "blind_sig", "sig":
                    b64e(sig_b)}))
                continue

            if t == "tss_recover_demo":
                which = req.get("which", [])
                shares = []
                for idx in which:
                    p = os.path.join(SHARE_DIR,
```

```
                        f"share_{int(idx)}.json")
73                  js = load_json(p, None)
74                  if js is None:
75                      raise ValueError("missing share file")
76                  shares.append((int(js["x"]), int(js["y"])))
77              master = shamir_combine(shares)
78              await ws.send(json.dumps({
79                  "type": "tss_ok",
80                  "master_sha256": b64e(sha256(master)),
81                  "expected_master_sha256": STATE["master_sha256"],
82                  "match": b64e(sha256(master)) ==
                        STATE["master_sha256"]
83              }))
84              continue
85
86          await ws.send(json.dumps({"type": "error", "error":
                "unknown type"}))
87
88      except Exception as e:
89          await ws.send(json.dumps({"type": "error", "error":
                str(e)}))
90
91
92  async def main():
93      print(f"[KMC] listening on ws://{HOST}:{PORT}")
94      async with websockets.serve(handler, HOST, PORT,
            max_size=2_000_000):
95          await asyncio.Future()
96
97
98  if __name__ == "__main__":
99      asyncio.run(main())
```

Листинг 3.4 – Файл `mrs.py`,

```
1  import json
2  import asyncio
3  import websockets
4
5  HOST = "127.0.0.1"
6  PORT = 9876
7
```

```
# simple in-memory      presence
CLIENTS = {}  # name -> websocket


async def handler(ws):
    """
    Client messages:
    - {"type":"hello","name":"A"}  (register connection on relay)
    - {"type":"relay","to":"B","payload":{...}} (forward payload)
    """
    name = None
    try:
        async for msg in ws:
            req = json.loads(msg)
            t = req.get("type")

            if t == "hello":
                name = req["name"]
                CLIENTS[name] = ws
                await ws.send(json.dumps({"type": "hello_ok"}))
                continue

            if t == "relay":
                to = req["to"]
                payload = req["payload"]
                if to not in CLIENTS:
                    await ws.send(json.dumps({"type": "error",
                       "error": f"{to} not connected"}))
                    continue
                await CLIENTS[to].send(json.dumps({"type": "inbox",
                   "from": name, "payload": payload}))
                await ws.send(json.dumps({"type": "relay_ok"}))
                continue

            await ws.send(json.dumps({"type": "error", "error":
               "unknown type"}))

    except websockets.ConnectionClosed:
        pass
    finally:
        if name and CLIENTS.get(name) is ws:
```

```
46                del CLIENTS[name]
47
48
49  async def main():
50      print(f"[MRS] listening on ws://{HOST}:{PORT}")
51      async with websockets.serve(handler, HOST, PORT,
            max_size=2_000_000):
52          await asyncio.Future()
53
54
55  if __name__ == "__main__":
56      asyncio.run(main())
```

4 Пример работы программы

KMS запускается как WebSocket-сервер и слушает соединения

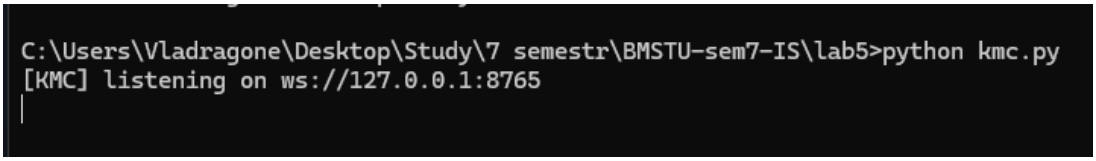

Рисунок 4.1 – KMS запуск

Далее запускаем MRS для пересылки сообщений и управлением клиентов:

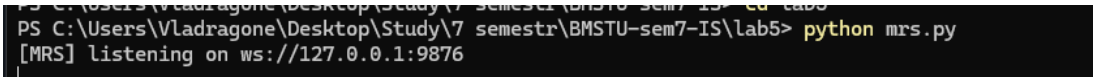

Рисунок 4.2 – MRS запуск

Далее регестрируем двух клиентов:

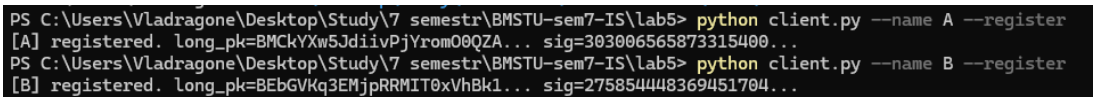

Рисунок 4.3 – Клиенты запуск

В результате регистрации появляются джсоны, которые хранят ключи этих клиентов:

Рисунок 4.4 – JSON для клиента A

Далее запускаем клиента B для прослушивания:

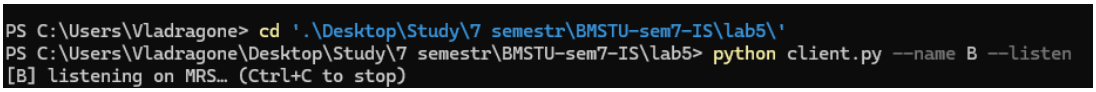

Рисунок 4.5 – Запуск клиента на прослушку

Далее отправляем сообщение с первого клиента:

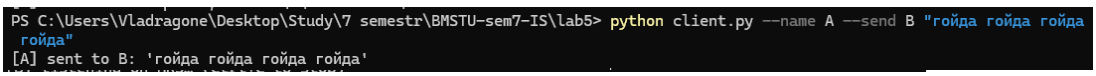

Рисунок 4.6 – Отправка сообщения с клиента A

В итоге получаем сообщение на клиенте B:

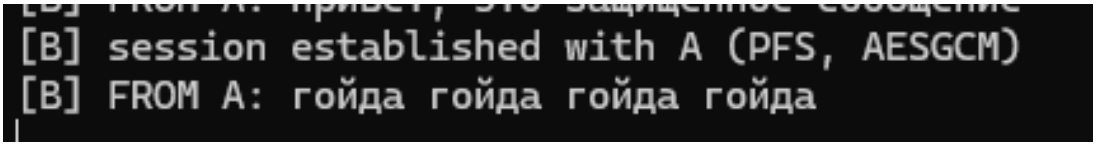

Рисунок 4.7 – Отправка сообщения с клиента A

При нагрузочном тестировании, при большом количестве клиентов заметно явный всплеск процессорной активности

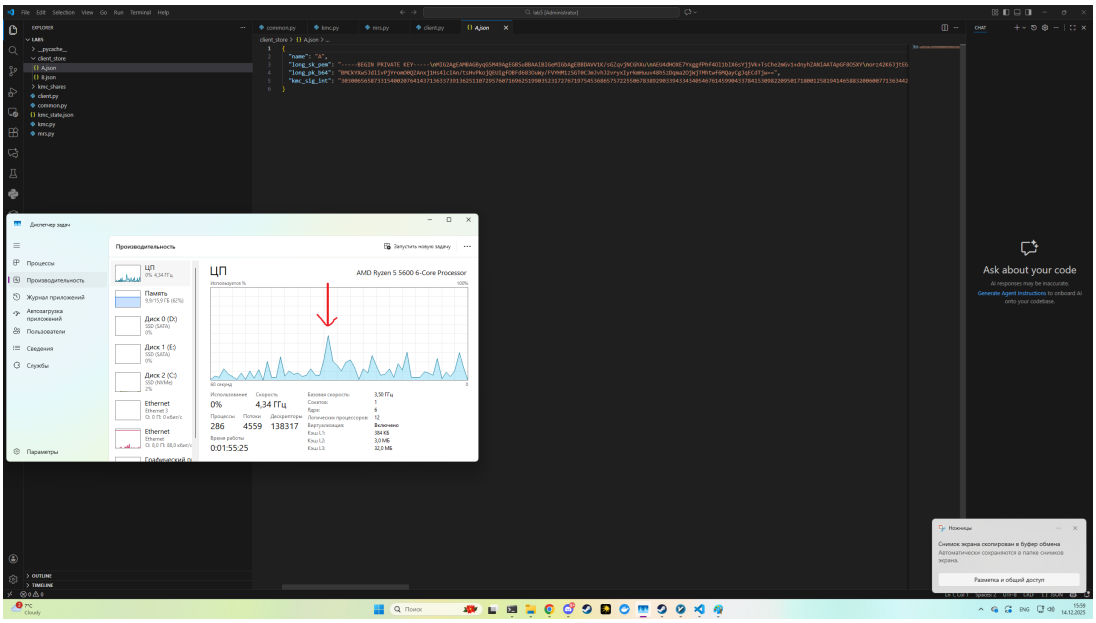

Рисунок 4.8 – Отправка сообщения с клиента A