Szyfrowanie plików z użyciem AES w trybie CFB

Autorzy: Mateusz Irski, Bartosz Gabruk, Paweł Szczeszek
Data: 25.03.2026

1. Krótki wstęp – charakterystyka trybu CFB

Projekt dotyczy implementacji aplikacji do szyfrowania i odszyfrowywania plików z użyciem algorytmu AES pracującego w trybie CFB (Cipher Feedback).

Tryb CFB zamienia szyfr blokowy w mechanizm działający podobnie do szyfru strumieniowego. Zamiast szyfrować kolejne bloki tekstu jawnego niezależnie, algorytm wykorzystuje poprzedni blok szyfrogramu jako wejście do wygenerowania kolejnego fragmentu strumienia szyfrującego.

Z punktu widzenia przetwarzania plików tryb CFB jest wygodny, ponieważ:

- nie wymaga dopełniania danych (paddingu),
- długość danych po szyfrowaniu pozostaje praktycznie taka sama jak przed szyfrowaniem, z uwzględnieniem nagłówka formatu,
- szyfrowanie i odszyfrowywanie mogą być realizowane strumieniowo,
- możliwe jest przetwarzanie dużych plików bez wczytywania ich w całości do pamięci,
- wymagane jest stosowanie unikalnego IV dla każdego szyfrowania wykonywanego tym samym kluczem.

2. Opis algorytmu

2.1. AES

AES (Advanced Encryption Standard) jest współczesnym szyfrem blokowym o rozmiarze bloku 128 bitów. Standard obsługuje trzy długości klucza:

- 128 bitów,
- 192 bity,
- 256 bitów.

Liczba rund zależy od długości klucza:

- AES-128: 10 rund,
- AES-192: 12 rund,
- AES-256: 14 rund.

W projekcie wykorzystano bibliotekę PyCryptodome, która realizuje wewnętrzne operacje AES zgodnie ze standardem. W implementacji przygotowano obsługę wszystkich trzech długości klucza i dodano benchmark pozwalający porównać ich praktyczną wydajność.

2.2. Tryb CFB

Działanie trybu CFB można opisać następująco:

1. Generowany jest losowy IV (wektor inicjalizujący).
2. IV zostaje zaszyfrowany algorytmem AES.
3. Wynik tej operacji jest łączony z danymi wejściowymi, tworząc szyfrogram.
4. Powstały blok szyfrogramu jest następnie używany do wygenerowania kolejnego fragmentu strumienia szyfrującego.
5. Przy odszyfrowywaniu wykorzystywany jest ten sam klucz i IV, a proces odzyskuje dane wejściowe blok po bloku.

W praktycznej implementacji projektu obsługą trybu CFB zarządza biblioteka PyCryptodome, natomiast kod aplikacji odpowiada za:

- przygotowanie klucza i IV,
- zapis nagłówka pliku,
- przetwarzanie danych porcjami,
- zapis i odczyt metadanych,
- benchmark dla trzech długości klucza,
- weryfikację poprawności odszyfrowania.

3. Najważniejsze fragmenty kodu wraz z objaśnieniami

3.1. Stałe silnika i zapis nagłówka pliku

Fragment z pliku `ssic/aes/engine.py` :

```
from Crypto.Cipher import AES

BLOCK_SIZE = AES.block_size  # 16
CHUNK_SIZE = 64 * 1024
MAGIC = b"AESC"
FORMAT_VERSION = 1

def _write_header(self, fout, mode, iv, file_size):
    fout.write(MAGIC)
    fout.write(
        struct.pack(
            "<BBBB",
            FORMAT_VERSION,
            mode.value,
            self._key_size.value,
            len(iv),
        )
    )
    fout.write(iv)
    fout.write(struct.pack("<Q", file_size))
```

Uwagi i objaśnienia:

- `BLOCK_SIZE` wynosi 16 bajtów, ponieważ AES operuje na blokach 128-bitowych.
- `CHUNK_SIZE = 64 * 1024` umożliwia przetwarzanie bardzo dużych plików bez konieczności wczytywania ich w całości do pamięci operacyjnej.
- `MAGIC = b"AESC"` identyfikuje własny format pliku szyfrowanego wykorzystywany w aplikacji.
- W nagłówku pliku zapisywane są wszystkie informacje potrzebne do późniejszego odszyfrowania:
  - wersja formatu,
  - tryb szyfrowania,
  - długość klucza,
  - długość IV,
  - IV,
  - oryginalny rozmiar pliku.

3.2. Tworzenie obiektu szyfrującego AES CFB

Fragment z pliku `ssic/aes/engine.py`:

```
def _make_cipher(self, iv: bytes | None = None) -> tuple[Any, bytes]:
    iv = iv if iv is not None else get_random_bytes(BLOCK_SIZE)
    cipher = AES.new(self._key, AES.MODE_CFB, iv=iv, segment_size=128)
    return cipher, iv
```

Uwagi i objaśnienia:

- Do pracy wykorzystywany jest tryb `AES.MODE_CFB`.
- `segment_size=128` oznacza wykorzystanie pełnego segmentu odpowiadającego rozmiarowi bloku AES.
- Jeżeli IV nie został podany, generowany jest losowo.
- Ten sam IV musi zostać zapisany w nagłówku pliku, aby możliwe było późniejsze poprawne odszyfrowanie danych.

3.3. Strumieniowe szyfrowanie pliku

Fragment z pliku `ssic/aes/engine.py`:

```
def _encrypt_stream(
    self,
    fin: BinaryIO,
    fout: BinaryIO,
    cipher: Any,
    file_size: int,
    progress_callback: Callable[[int, int], None] | None,
) -> None:
    processed = 0
    while True:
        chunk = fin.read(CHUNK_SIZE)
        if not chunk:
            break
        fout.write(cipher.encrypt(chunk))
        processed += len(chunk)
        callback = cast(Callable[[int, int], None] | None, progress_callback)
        if callback is not None:
            callback(processed, file_size)
```

Uwagi i objaśnienia:

- Dane wejściowe są czytane w porcjach o stałym rozmiarze.
- Program nie musi przechowywać całego pliku w pamięci, co ma kluczowe znaczenie przy plikach rzędu 10 GB.
- Taki sposób działania pozwala zachować stabilne zużycie pamięci niezależnie od wielkości pliku.
- Analogiczna logika została zastosowana przy odszyfrowywaniu.

3.4. Zapis metadanych szyfrowania

Fragment z pliku `ssic/aes/metadata.py`:

```
metadata = AESEncryptionMetadata(
    format="ssic-aes-metadata/v1",
    created_at=datetime.now(UTC).isoformat(),
    source_path=str(src),
    encrypted_path=str(encrypted_path),
    mode=mode.name,
    key_size=key_size.label,
    key_hex=key.hex(),
    file_size_bytes=file_size_bytes,
)
metadata_path.write_text(json.dumps(asdict(metadata), indent=2), encoding="utf-8")
```

Uwagi i objaśnienia:

- Metadane są zapisywane do osobnego pliku JSON obok szyfrogramu.
- Plik ten zawiera:
    - ścieżkę do pliku źródłowego,
    - ścieżkę do pliku zaszyfrowanego,
    - tryb pracy,
    - długość klucza,
    - klucz w postaci heksadecymalnej,
    - rozmiar pliku.
- Rozwiązanie to upraszcza odszyfrowywanie oraz testowanie aplikacji.
- Należy jednak zaznaczyć, że przechowywanie klucza obok zaszyfrowanych danych jest rozwiązaniem wyłącznie dydaktycznym i nie powinno być stosowane w systemach produkcyjnych.

3.5. Benchmark dla trzech długości klucza

Fragment z pliku `ssic/tui/screens/aes_benchmark.py`:

```
 key_sizes = [AESKeySize.AES_128, AESKeySize.AES_192, AESKeySize.AES_256]
mode = AESCipherMode.CFB

for ks in key_sizes:
    enc_path = temp_base / f"bench_{ks.bits}.aes"
    dec_path = temp_base / f"bench_{ks.bits}.dec"

    engine = AESEngine(key_size=ks)

    t0 = time.perf_counter()
    engine.encrypt_file(src_path, enc_path, mode)
    enc_time = time.perf_counter() - t0

    t0 = time.perf_counter()
    engine.decrypt_file(enc_path, dec_path)
    dec_time = time.perf_counter() - t0
```

Uwagi i objaśnienia:

- Benchmark automatycznie testuje trzy wersje AES:
  - AES-128,
  - AES-192,
  - AES-256.
- Dla każdej wersji mierzony jest oddzielnie czas szyfrowania oraz czas odszyfrowywania.
- Dzięki temu możliwe jest bezpośrednie porównanie wydajności dla różnych długości klucza.

3.6. Weryfikacja poprawności odszyfrowania

Fragment z pliku `ssic/crypto/hash_utils.py`:

```
def file_sha256(path, *, progress=None):
    digest = hashlib.sha256()
    total = path.stat().st_size
    processed = 0
    with open(path, "rb") as handle:
        while True:
            chunk = handle.read(CHUNK_SIZE)
            if not chunk:
                break
            digest.update(chunk)
            processed += len(chunk)
            if progress is not None:
                progress(processed, total)
    return digest.hexdigest()

def compare_files(left_path, right_path):
    return FileComparisonResult(
        left_path=left_path,
        right_path=right_path,
        left_sha256=file_sha256(left_path),
        right_sha256=file_sha256(right_path),
    )
```

Uwagi i objaśnienia:

- SHA-256 służy do potwierdzenia, że odszyfrowany plik jest zgodny z plikiem oryginalnym.
- Porównanie skrótów jest wygodnym i czytelnym sposobem weryfikacji poprawności działania programu.
- Podobnie jak szyfrowanie, także obliczanie hashy odbywa się porcjami, dzięki czemu działa poprawnie również dla bardzo dużych plików.

4. Eksperyment

4.1. Cel eksperymentu

Celem eksperymentu było sprawdzenie:

- czy implementacja poprawnie szyfruje plik o rozmiarze co najmniej 10 GB,
- czy odszyfrowany plik jest zgodny z oryginałem,
- jak zmienia się czas szyfrowania i odszyfrowywania dla trzech długości klucza AES.

4.2. Dane testowe

W eksperymencie wykorzystano plik wejściowy:

- plik źródłowy: `D:\SSIC\SSiC\generated_files\random_20260325_162021.bin`
- rozmiar pliku: `10 485 760 000` bajtów

Przykład szyfrowania zapisany w metadanych:


```
{
  "format": "ssic-aes-metadata/v1",
  "created_at": "2026-03-25T15:32:07.973402+00:00",
  "source_path": "D:\\SSIC\\SSiC\\generated_files\\random_20260325_162021.bin",
  "encrypted_path": "D:\\SSIC\\SSiC\\generated_files\\zaszyfrowany_10gb.aes",
  "mode": "CFB",
  "key_size": "AES-128",
  "key_hex": "a55f0cd1349285dc10b23a792d6b2075",
  "file_size_bytes": 10485760000
}
```


Na podstawie powyższych metadanych można stwierdzić, że:

- zastosowano tryb `CFB`,
- wykorzystano klucz `AES-128`,
- zaszyfrowany plik został zapisany jako `zaszyfrowany_10gb.aes`.

4.3. Konfiguracja sprzętowa

Eksperyment został przeprowadzony na komputerze o następującej konfiguracji:

- procesor: `Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz`
- pamięć RAM: `16 GB`
- system operacyjny: `Windows 10`
- interpreter: `Python 3.14.0`

4.4. Wyniki

Rozmiar pliku testowego według benchmarku:

Wyniki pomiarów:

Wariant	Czas szyfrowania	Prędkość szyfrowania	Czas odszyfrowywania	Prędkość odszyfrowywania
AES-128	56.163 s	178.1 MB/s	37.209 s	268.7 MB/s
AES-192	54.786 s	182.5 MB/s	37.811 s	264.5 MB/s
AES-256	89.051 s	112.3 MB/s	41.630 s	240.2 MB/s

4.5. Analiza wyników

Najważniejsze obserwacje:

- Implementacja poprawnie obsługuje szyfrowanie i odszyfrowywanie dużych plików.
- Strumieniowe przetwarzanie danych pozwala utrzymać stabilne zużycie pamięci nawet dla pliku około 10 GB.
- AES-128 i AES-192 osiągnęły bardzo zbliżone czasy działania.
- AES-256 okazał się wyraźnie wolniejszy podczas szyfrowania, co jest zgodne z większą liczbą rund algorytmu.
- Odszyfrowywanie było szybsze niż szyfrowanie w wykonanych pomiarach.
- Wyniki pokazują praktyczny kompromis pomiędzy poziomem bezpieczeństwa a wydajnością.

5. Wnioski

- Tryb CFB dobrze nadaje się do szyfrowania plików, ponieważ nie wymaga paddingu i umożliwia wygodne przetwarzanie strumieniowe, co upraszcza pracę z plikami binarnymi o dowolnym rozmiarze.
- CFB łączy cechy szyfru blokowego i strumieniowego: AES nadal pracuje na blokach 128-bitowych, ale z punktu widzenia aplikacji dane mogą być przetwarzane kolejno, bez konieczności wcześniejszego dzielenia pliku na niezależne porcje logiczne.
- Istotną właściwością trybu CFB jest zależność kolejnych fragmentów szyfrogramu od poprzednich, co oznacza, że poprawny dobór i zapis IV ma kluczowe znaczenie dla poprawnego odszyfrowania danych.
- W CFB błąd w jednym bloku: psuje ten blok + część następnego, ale nie cały plik.
- AES-128 i AES-192 zapewniły w badaniu bardzo zbliżoną wydajność, natomiast AES-256 był zauważalnie wolniejszy, co wynika z większej liczby rund wykonywanych przez algorytm.
- AES-256 zapewnia najwyższy poziom bezpieczeństwa spośród testowanych wariantów, ale kosztem niższej wydajności, dlatego wybór długości klucza powinien zależeć od wymagań bezpieczeństwa i czasu przetwarzania.

- Benchmark dla trzech długości klucza umożliwia łatwe porównanie praktycznych różnic pomiędzy AES-128, AES-192 i AES-256 w rzeczywistym scenariuszu pracy na dużym pliku.
- Zapis metadanych znacząco upraszcza pracę z aplikacją, ale przechowywanie klucza obok danych należy traktować wyłącznie jako rozwiązanie dydaktyczne, a nie jako praktykę dopuszczalną w systemie produkcyjnym.