Szyfrowanie plików z użyciem 3DES w trybie OFB

Autorzy: Mateusz Irski, Bartosz Gabruk, Paweł Szczeszek **Data:** 19.03.2026

1. Krótki wstęp – charakterystyka trybu OFB

Projekt dotyczy implementacji aplikacji do szyfrowania i odszyfrowywania plików z użyciem algorytmu 3DES pracującego w trybie OFB (Output Feedback). Tryb OFB zamienia szyfr blokowy w szyfr działający podobnie do szyfru strumieniowego: zamiast szyfrować bezpośrednio kolejne bloki danych wejściowych, generowany jest strumień klucza, który następnie łączony jest z danymi operacją XOR.

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

- nie wymaga dopełniania danych (paddingu),
- długość danych po szyfrowaniu pozostaje praktycznie taka sama jak przed szyfrowaniem,
- szyfrowanie i odszyfrowywanie korzystają z tego samego mechanizmu generowania strumienia klucza,
- pojedynczy błąd w szyfrogramie wpływa zwykle tylko na odpowiadający mu fragment tekstu jawnego,
- konieczne jest jednak stosowanie unikalnego IV dla każdego szyfrowania wykonywanego tym samym kluczem.

2. Zwięzły opis algorytmu

2.1. 3DES

3DES (Triple DES) jest rozwinięciem algorytmu DES. Operacja szyfrowania wykonywana jest trzykrotnie, co zwiększa bezpieczeństwo względem pojedynczego DES. W projekcie wykorzystywany jest klucz o długości 24 bajtów, zgodny z wymaganiami biblioteki PyCryptodome dla DES3.

2.2. Tryb OFB

Działanie OFB można opisać następująco:

1. Generowany jest losowy IV (wektor inicjalizujący).
2. IV zostaje zaszyfrowany algorytmem 3DES, tworząc pierwszy blok strumienia klucza.
3. Ten blok jest łączony operacją XOR z danymi wejściowymi, dając szyfrogram.
4. Kolejny blok strumienia klucza powstaje przez ponowne zaszyfrowanie poprzedniego bloku strumienia.
5. Przy odszyfrowaniu wykorzystywany jest dokładnie ten sam strumień klucza, dlatego proces jest symetryczny.

W praktycznej implementacji projektu generowaniem strumienia OFB zarządza biblioteka PyCryptodome, natomiast kod aplikacji odpowiada za:

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

3. Najważniejsze fragmenty kodu wraz z objaśnieniami

3.1. Stałe silnika i zapis nagłówka pliku

Fragment z pliku ssic/crypto/engine.py:

```
from Crypto.Cipher import DES3

BLOCK_SIZE = DES3.block_size  # 8
KEY_SIZE = 24
CHUNK_SIZE = 64 * 1024
MAGIC = b"3DSC"
FORMAT_VERSION = 1

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

Uwagi i objaśnienia:

- BLOCK_SIZE wynosi 8 bajtów, ponieważ 3DES operuje na blokach 64-bitowych.
- KEY_SIZE = 24 oznacza użycie pełnego klucza 3DES.
- CHUNK_SIZE = 64 * 1024 pozwala przetwarzać nawet duże pliki bez wczytywania ich w całości do pamięci.
- Nagłówek zapisuje wszystkie informacje potrzebne do późniejszego odszyfrowania pliku.

3.2. Strumieniowe szyfrowanie pliku

Fragment z pliku ssic/crypto/engine.py:

```
def _encrypt_stream(self, fin, fout, cipher, file_size, progress_callback):
    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 są czytane i przetwarzane porcjami o stałym rozmiarze.
- Zużycie pamięci nie zależy więc bezpośrednio od wielkości pliku.
- Taki sposób działania jest szczególnie istotny dla dużych plików binarnych.

Analogiczna logika została zastosowana przy odszyfrowywaniu, dzięki czemu oba procesy mają podobną złożoność i charakterystykę wydajnościową.

3.3. Zapis metadanych szyfrowania

Fragment z pliku ssic/crypto/metadata.py:

```
metadata = EncryptionMetadata(
    format="ssic-3des-metadata/v1",
    created_at=datetime.now(UTC).isoformat(),
    source_path=str(src),
    encrypted_path=str(encrypted_path),
    mode=mode.name,
    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.
- Dzięki temu podczas odszyfrowywania można odzyskać informacje o trybie i kluczu.
- Rozwiązanie to jest wygodne w projekcie laboratoryjnym, ponieważ upraszcza testowanie.
- Jednocześnie należy zaznaczyć, że zapis klucza w jawnej postaci (key_hex) nie jest rozwiązaniem bezpiecznym produkcyjnie. W praktycznych systemach klucz nie powinien być przechowywany obok zaszyfrowanych danych.

3.4. Obliczanie i porównywanie skrótu SHA-256

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 tutaj do potwierdzenia, że plik odszyfrowany jest identyczny z oryginałem.
- Porównanie na podstawie skrótu jest wygodne i czytelne z punktu widzenia raportowania wyników.
- Podobnie jak w przypadku szyfrowania, plik jest czytany porcjami, więc narzędzie działa również dla większych plików.

4. Eksperyment

4.1. Cel eksperymentu

Celem eksperymentu było sprawdzenie:

1. czy implementacja poprawnie szyfruje i odszyfrowuje pliki,
2. czy odszyfrowany plik jest identyczny z oryginałem.

4.2. Wyniki

Rozmiar pliku - 2GB Czas szyfrowania - 3m15s Czas deszyfrowania - 3m16s

Konfiguracja:

- Ryzen 7 5700U
- 16GB RAM

4.3. Analiza wyników

Najważniejsze obserwacje:

- czasy szyfrowania i odszyfrowywania jest bardzo zbliżony, co jest zgodne z charakterem trybu OFB,
- porównanie plików za pomocą SHA-256 jest wyraźnie szybsze niż samo szyfrowanie 3DES.

Wyniki potwierdzają, że implementacja działa poprawnie i zachowuje przewidywalne właściwości wydajnościowe dla przetwarzania strumieniowego.

5. Podsumowanie

W projekcie zrealizowano kompletne narzędzie do pracy z plikami szyfrowanymi przy użyciu 3DES w trybie OFB. Aplikacja pozwala:

- generować pliki testowe,
- szyfrować dane do własnego formatu plikowego,
- zapisywać metadane potrzebne do późniejszego odczytu,
- odszyfrowywać pliki,
- porównywać pliki na podstawie SHA-256.

Najważniejszą cechą implementacji jest strumieniowe przetwarzanie danych, dzięki któremu program nadaje się także do pracy na większych plikach. Rozdzielenie logiki kryptograficznej, logiki pomocniczej i warstwy TUI poprawia czytelność projektu oraz ułatwia dalszy rozwój.

6. Wnioski

1. Tryb OFB dobrze nadaje się do szyfrowania plików, ponieważ nie wymaga paddingu i umożliwia wygodne przetwarzanie strumieniowe.
2. Zastosowanie stałego rozmiaru porcji (CHUNK_SIZE) pozwala utrzymać stabilne zużycie pamięci.
3. Zapis metadanych znacząco upraszcza pracę z aplikacją, ale przechowywanie klucza obok danych należy traktować wyłącznie jako rozwiązanie dydaktyczne.