
SHANNON W. BRAY
IMPLEMENTACJA KRYPTOGRAFII W PYTHONIE
WPROWADZENIE
Helion
WILEY

Tytuł oryginału: Implementing Cryptography Using Python

Tłumaczenie: Filip Kamiński

ISBN: 978-83-283-7729-5

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/imkrpy
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/imkrpy.zip

Printed in Poland.

Spis treści

O autorze

Shannon W. Bray rozpoczął swoją karierę w branży IT w 1997 r. po odejściu ze służby w marynarce wojennej Stanów Zjednoczonych. Na początku pracował nad kontrolerami PLC w rejonie Zatoki Meksykańskiej. Następnie, aż do pojawienia się platformy .NET w 2000 r., tworzył aplikacje biznesowe w różnych technologiach. Od 2000 r. preferowanymi przez Shannona językami programowania są C++, C#, Windows PowerShell i Python. Kontynuował karierę programisty aż do momentu, w którym zaczął pracować z Microsoft SharePoint. Stos technologiczny Microsoftu skłonił Shannona do odejścia od pisania samego kodu i zajęcia się tworzeniem dużych bezpiecznych rozwiązań IT. Podejmując współpracę z wieloma agencjami rządu USA, Shannon angażował się w projekty na całym świecie. Kryptografią zainteresował się podczas studiów magisterskich z cyberbezpieczeństwa, a obecnie kontynuuje badania nad kryptografią i cyberbezpieczeństwem w ramach studiów doktoranckich z informatyki.

Shannon posiada certyfikaty Microsoft Certified Master, Microsoft Certified Solutions Master, Certified Information Security Manager (CISM), Security+ (Plus), CompTIA Advanced Security Practitioner (CASP+), a także wiele innych certyfikatów branżowych. Oprócz pisania występuje na krajowych kongresach technologicznych i pracuje jako mentor prowadzący zajęcia dla młodzieży.

Shannon mieszka w Stanach Zjednoczonych, w pobliżu Raleigh w Karolinie Północnej, wraz z żoną Stephanie, dwiema córkami Eden i Kenną oraz synem Haydenem. W sezonie lubi nurkować u wybrzeży Karoliny Północnej. Przez pozostałą część roku pracuje nad domowymi projektami technologicznymi. Wykorzystuje w nich komputery jednopłytkowe i bardzo często śmigła. Obecnie tworzy domowy system bezpieczeństwa oparty na Pythonie, kryptografii i dronach. W 2020 r. kandydował do Senatu USA w Karolinie Północnej. Jego celami były wprowadzenie problemów cyberbezpieczeństwa do głównego nurtu i pomoc w zrozumieniu znaczenia pełnego szyfrowania (ang. *end-to-end encryption*).

Z Shannonem można się skontaktować za pomocą serwisu LinkedIn. Jego profil znajdziesz pod adresem *www.linkedin.com/in/shannonbray*.

Rozdział

6

Kryptografia wizualna

W tym momencie na pewno już umiesz szyfrować i odszyfrowywać wiadomości. Niniejszy rozdział pozwoli Ci poszerzyć wiedzę o kryptografię obrazów. Dzięki zaprezentowanym w nim ćwiczeniom zidentyfikujesz problemy związane z różnymi trybami szyfrowania, a także nauczysz się alternatywnych sposobów szyfrowania. Omówienie kryptografii wizualnej zakończy steganografia.

W tym rozdziale:

- Poznasz biblioteki graficzne i kryptograficzne.
- Dowiesz się więcej o trybach kodowania w szyfrowaniu AES.
- Poznasz różne metody kryptografii wizualnej.
- Z pomocą Pythona ukryjesz i odczytasz multimedia w obrazie.

Prosty przykład

Zanim przejdę do omówienia bibliotek, które można wykorzystać w kryptografii wizualnej, chcę pokazać prostą metodę, która pozwoli Ci zaszyfrować i odszyfrować plik. Cały proces polega na przechodzeniu przez bity obrazu w pętli i równoczesnym obliczaniu alternatywy rozłącznej (XOR) bitów z kluczem. Aby odszyfrować obraz, wystarczy odwrócić operację XOR.

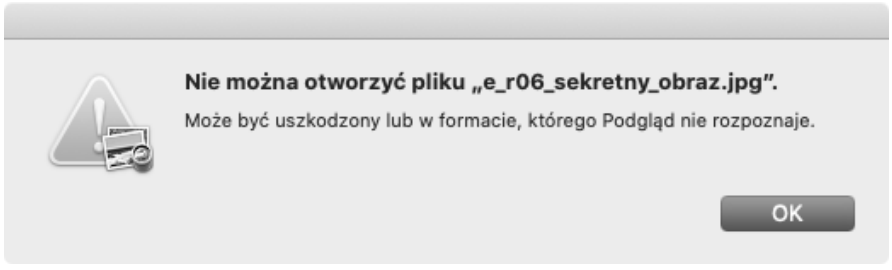
W tym przykładzie, a także w kolejnych w niniejszym rozdziale, zaszyfrowany obraz zapiszę z przedrostkiem *e* w nazwie. Po odszyfrowaniu literę *e* zastąpię literą *d*. Powodem, dla którego stosuję tę konwencję, jest możliwość porównania oryginalnego obrazu z odszyfrowanym, tak aby sprawdzić, czy nie wystąpiła widoczna utrata danych.

Rysunek 6.1 przedstawia obraz, który będę wykorzystywał w tym rozdziale. Zamiast nim możesz się posłużyć jakimkolwiek innym obrazkiem. Ten dość mały plik będzie jednak wystarczający, dopóki nie dojdziemy do końca rozdziału. Do ukrycia dużych obiektów binarnych (ang. *blob*) może być jednak potrzebny znacznie większy plik. Czas na krótki kod.

Rysunek 6.1. Oryginalny plik r06_sekretny_obraz.jpg

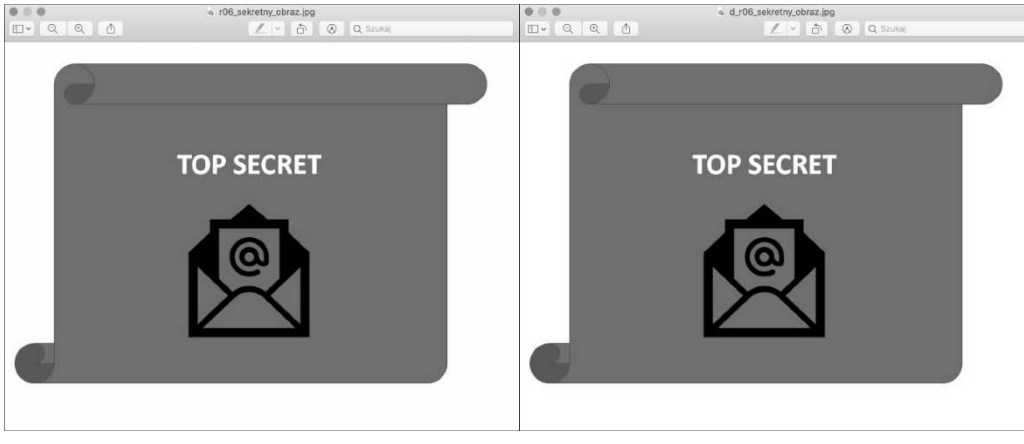
```
print('Program poszukuje pliku: r06_sekretny_obraz.jpg')
fo = open("./r06_sekretny_obraz.jpg", "rb")
image = fo.read()
fo.close()
print()
print("Klucz: 42.")
image = bytearray(image)
key = 42
for index, value in enumerate(image):
    image[index] = value^key

print()
print('Obraz został zaszyfrowany. Sprawadź plik e_r06_sekretny_obraz.jpg')

fo = open("./e_r06_sekretny_obraz.jpg", "wb")
fo.write(image)
fo.close()
image = bytearray(image)
for index, value in enumerate(image):
    image[index] = key^value

print()
print('Obraz został odszyfrowany. Sprawadź plik d_r06_sekretny_obraz.jpg')

fo = open("./d_r06_sekretny_obraz.jpg", "wb")
fo.write(image)
fo.close()
```

Obraz jest teraz nieczytelny, a jego otwarcie, w zależności od używanego oprogramowania, może powodować wyświetlenie błędu (zobacz rysunek 6.2). Nie jest to jednak problem związany z wyświetlaniem. Nadal możesz wysłać plik w dowolne miejsce na świecie, a przy tym nie musisz się martwić, że ktoś odszyfruje go bez klucza, którym w tym przypadku jest liczba 42.



Rysunek 6.2. Komunikat o błędzie otwarcia pliku e_r06_sekretny_obraz.jpg

Odszyfrowanie odtworzy obraz bez utraty danych, tak jak pokazałem na rysunku 6.3. Kwestia utraty danych jest istotna, ponieważ użycie niektórych trybów będzie się wiązało z pewnymi stratami. Obraz po lewej stronie to oryginał, obraz po prawej został zaszyfrowany, a następnie odszyfrowany. W dalszej części rozdziału przedstawię tryby szyfrowania, które działają dobrze w przypadku zwykłych plików, ale nie sprawdzają się w przypadku obrazów. Jednym z takich przykładów jest omówiony już krótko tryb ECB.



Rysunek 6.3. Dwa obrazki obok siebie: r06_sekretny_obraz.jpg i d_r06_sekretny_obraz.jpg

W następnym podrozdziale przyjrzymy się bardziej złożonym rozwiązaniom, które wykorzystują biblioteki kryptograficzne.

Biblioteki graficzne i steganograficzne

Omówię teraz biblioteki, które ułatwią pracę z kryptografią. Powrócę na chwilę do wprowadzonej w rozdziale 1. biblioteki *cryptography* i przedstawię bibliotekę *cryptosteganography*, która pozwoli Ci przeanalizować pokazane w dalszej części tego rozdziału przykłady steganografii.

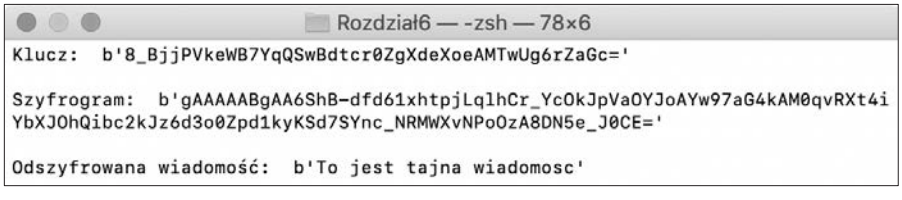
Biblioteka cryptography

Pakiet *cryptography* to zbiór metod i prymitywów kryptograficznych. Biblioteka obsługuje różne wersje Pythona i szczyci się mianem „kryptograficznej biblioteki standardowej". Możesz ją zainstalować za pomocą polecenia `pip install cryptography`.

Biblioteka *cryptography* zawiera funkcje wysokiego poziomu i niskopoziomowe interfejsy do typowych algorytmów kryptograficznych, takich jak szyfrowanie symetryczne, skróty i funkcje wyprowadzania kluczy. Poniższy przykład szyfrowania szyfrem Ferneta korzysta z interfejsu wysokiego poziomu. Uruchom następujący kod, aby upewnić się, że na Twoim komputerze została zainstalowana biblioteka *cryptography*:

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
f = Fernet(key)
print("Klucz: ", key)
print()
ciphertext = f.encrypt(b"To jest tajna wiadomosc")
print("Szyfrogram: ", ciphertext)
print()
plaintext = f.decrypt(ciphertext)
print("Odszyfrowana wiadomość: ", plaintext)
```

Jeśli biblioteka *cryptography* została zainstalowana poprawnie, to możesz teraz wygenerować nowy klucz oraz zaszyfrować i odszyfrować wiadomość `'To jest tajna wiadomosc'`. Wynik powinien przypominać ten z rysunku 6.4. Jedyną różnicą będzie klucz, który jest unikatowy.



Rysunek 6.4. Test biblioteki cryptography

Biblioteka cryptosteganography

Steganografia to sztuka ukrywania informacji w różnego rodzaju obiektach multimedialnych, takich jak obrazy czy pliki audio, w taki sposób, że nikt poza nadawcą i odbiorcą nie podejrzewa, iż wiadomość w ogóle istnieje. Steganografia jest przykładem zapewniania bezpieczeństwa przez niejawność (ang. *security through obscurity*). W tym rozdziale przedstawię nie tylko metody ukrywania jawnego tekstu w obrazach, ale również sposoby na ukrycie zaszyfrowanych danych multimedialnych. Niezaszyfrowane dane ukryte w obrazach można wyekstrahować za pomocą wielu dostępnych na licencji freeware narzędzi. Dlatego przed ich ukryciem w obrazie możesz je zaszyfrować.

Drugą biblioteką, którą zamierzam przedstawić, jest *cryptosteganography*. Moduł ten umożliwia przechowywanie wiadomości lub plików chronionych za pomocą algorytmu AES-256 wewnątrz obrazów. Instalację biblioteki w wirtualnym środowisku lub projekcie możesz przeprowadzić za pomocą narzędzia `pip3`. Biblioteka *cryptosteganography* została stworzona z myślą o Pythonie w wersji 3 i w nowszych.

Kryptografia wizualna

W tym podrozdziale przedstawię narzędzia i formaty związane z kryptografią wizualną. Rozpocznę od szyfrowania plików za pomocą algorytmu Ferneta. Przykład ten pozwoli Ci szyfrować i odszyfrowywać obrazy przy użyciu tego algorytmu. W dalszej części podrozdziału zbadamy, jak tryb kodowania wpływa na obrazy zaszyfrowane za pomocą algorytmu AES.

Szyfrowanie zawartości pliku za pomocą algorytmu Ferneta

W tym punkcie omówię szyfrowanie obrazów za pomocą algorytmu Ferneta. Algorytm ten jest przykładem szyfrowania symetrycznego z uwierzytelnianiem (ang. *authenticated symmetric encryption*). Jak być może pamiętasz, szyfrowanie symetryczne to rodzaj szyfrowania, w którym zarówno do szyfrowania, jak i odszyfrowywania stosuje się ten sam klucz. Zacznę od wygenerowania klucza, a następnie zapiszę go w pliku. Każde wywołanie funkcji `gene_key()` powoduje wygenerowanie nowego klucza. Zgubienie klucza uniemożliwi Ci odszyfrowanie zaszyfrowanych nim danych. Dlatego po wygenerowaniu klucz zapiszę na dysku. Pamiętaj, że po wygenerowaniu klucza zawsze należy go zapisać. Alternatywą jest stworzenie swojego klucza.

Kod pozwalający wygenerować i zapisać klucz wygląda następująco:

```
key = Fernet.generate_key()
with open("r06.key", "wb") as key_file:
key_file.write(key)
```

Po zapisaniu do załadowania pliku z kluczem możesz wykorzystać następujący kod:

```
return open("r06.key", "rb").read()
```

Po przygotowaniu algorytmu szyfrowania i klucza możesz zaszyfrować wiadomość. Nie zapomnij też o jej zakodowaniu. Kodowanie spowoduje konwersję łańcucha znaków na bajty, które można następnie zaszyfrować. Przypisanie wiadomości do zmiennej powinno przebiegać w następujący sposób:

```
plaintext = "Mam pewien sekret, ktorym podziele sie tylko z osoba znajaca klucz.".encode()
```

Teraz można rozpocząć proces szyfrowania:

```
f = Fernet(key)
ciphertext = f.encrypt(plaintext)
print(ciphertext)
```

Następnie załaduję plik zawierający tekst jawny i zaszyfruję go za pomocą metody `encrypt()`. Zaszyfrowany tekst zapiszę następnie na dysku:

```
filename = "plik.txt"
f = Fernet(key)
with open(filename, "rb") as file:
# Wczytanie zawartości pliku
    file_data = file.read()
# Szyfrowanie danych
encrypted_data = f.encrypt(file_data)
# Zapis zaszyfrowanego pliku
with open(filename, "wb") as file:
    file.write(encrypted_data)
```

Proces odszyfrowywania przebiega prawie identycznie. Stosuje się w nim metodę `decrypt()` obiektu `Fernet`:

```
$filename = "plik.txt"
f = Fernet(key)
with open(filename, "rb") as file:
# Wczytanie zaszyfrowanej zawartości pliku
encrypted_data = file.read()
# Odszyfrowywanie danych
decrypted_data = f.decrypt(encrypted_data)
# Odtworzenie oryginalnego pliku
with open(filename, "wb") as file:
    file.write(decrypted_data)
```

Teraz, gdy znasz już koncepcje, wszystkie powyższe fragmenty możesz połączyć w coś bardziej elastycznego. Oto skrypt, który przyjmuje klucz, plik i flagę wskazującą, czy plik ma zostać zaszyfrowany, czy odszyfrowany:

```
from cryptography.fernet import Fernet
import os

def write_key():
"""
    Generowanie i zapis klucza do pliku
"""
    key = Fernet.generate_key()
    with open("klucz.key", "wb") as key_file:
        key_file.write(key)

def load_key():
"""
    Wczytanie pliku o nazwie `klucz.key` z bieżącego katalogu
"""
    return open("klucz.key", "rb").read()

def encrypt(filename, key):
"""
    Szyfrowanie i zapis pliku o nazwie filename (str) kluczem key (bytes)
"""
    f = Fernet(key)
    with open(filename, "rb") as file:
# Wczytanie zawartości pliku
        file_data = file.read()
# Szyfrowanie danych
    encrypted_data = f.encrypt(file_data)
# Zapis zaszyfrowanego pliku
with open(filename, "wb") as file:
        file.write(encrypted_data)

def decrypt(filename, key):
"""
    Odszyfrowywanie i zapis pliku o nazwie filename (str). Plik zaszyfrowno kluczem key (bytes)
"""
    f = Fernet(key)
    with open(filename, "rb") as file:
# Wczytanie zaszyfrowanej zawartości pliku
encrypted_data = file.read()
```

```
# Odszyfrowywanie danych
    decrypted_data = f.decrypt(encrypted_data)
# Odtworzenie oryginalnego pliku
with open(filename, "wb") as file:
        file.write(decrypted_data)


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="Prosty skrypt szyfrujący pliki.")
parser.add_argument("file", help="Nazwa pliku do zaszyfrowania/odszyfrowania.")
parser.add_argument("-g", "--generate-key", dest="generate_key", action="store_true",
help="Flaga określająca, czy klucz ma zostać wygenerowany (jeżeli nie, to użyty zostanie
↪istniejący klucz).")
parser.add_argument("-e", "--encrypt", action="store_true",
help="Szyfrowanie. Dozwolone opcje to tylko -e i -d.")
parser.add_argument("-d", "--decrypt", action="store_true",
help="Szyfrowanie. Dozwolone opcje to tylko -e i -d.")

args = parser.parse_args()
    file = args.file
    generate_key = args.generate_key

    if generate_key:
write_key()
    # Wczytywanie klucza
key = load_key()

    encrypt_ = args.encrypt
    decrypt_ = args.decrypt

    if encrypt_ and decrypt_:
raise TypeError("Określ tryb pracy (szyfrowanie/odszyfrowywanie).")
elif encrypt_:
        encrypt(file, key)
    elif decrypt_:
        decrypt(file, key)
else:
        raise TypeError("Określ tryb pracy (szyfrowanie/odszyfrowywanie).")
```

Szyfrowanie obrazu za pomocą algorytmu Ferneta

Korzystając z kodu z poprzedniego punktu, możesz teraz stworzyć skrypt, który pozwoli Ci zaszyfrować określony obraz i zapisać go w pliku o ustalonej nazwie. W tym przypadku plik *r06_sekretny_obraz.jpg* zostanie zaszyfrowany i zapisany jako *e_r06_sekretny_obraz.jpg*, a następnie natychmiast odszyfrowany i zapisany jako *d_r06_sekretny_obraz.jpg*. Rozdzielczości oryginalnego i odszyfrowanego obrazu będą takie same, ponieważ algorytm Ferneta działa na samym pliku i nie zmienia pikseli w obrazie.

Na poniższym listingu brakuje funkcji wczytującej i generującej klucz, ponieważ procesy te odbywają się tak samo jak w poprzednim przykładzie. Pokazany tu proces szyfrowania nie ogranicza się do obrazów i można go stosować z danymi różnych typów. Wynikiem działania tego skryptu powinno być pełne szyfrowanie na poziomie pliku, a nie na poziomie pikseli:

```
def encrypt(filename, newfile, key):
"""
    Szyfrowanie obrazu filename (str) kluczem key (bytes). Zapis wyniku w pliku newfile (str).
"""
    f = Fernet(key)
    with open(filename, "rb") as file:
# Wczytanie zawartości pliku
        file_data = file.read()
# Szyfrowanie danych
    encrypted_data = f.encrypt(file_data)
# Zapis zaszyfrowanego pliku
with open(newfile, "wb") as file:
        file.write(encrypted_data)

def decrypt(filename, newfile, key):
"""
    Odszyfrowywanie obrazu filename (str), który zaszyfrowano kluczem key (bytes). Zapis
    ↪wyniku w pliku newfile (str).
"""
    f = Fernet(key)
    with open(filename, "rb") as file:
# Wczytanie zaszyfrowanej zawartości pliku
encrypted_data = file.read()
# Odszyfrowywanie danych
    decrypted_data = f.decrypt(encrypted_data)
# Odtworzenie oryginalnego pliku
with open(newfile, "wb") as file:
        file.write(decrypted_data)

key = Fernet.generate_key()

enc = encrypt("./r06_sekretny_obraz.jpg", "./e_r06_sekretny_obraz.jpg", key)
dec = decrypt("./e_r06_sekretny_obraz.jpg", "./d_r06_sekretny_obraz.jpg", key)
```

AES i tryby kodowania

AES (ang. *Advanced Encryption Standard* — zaawansowany standard szyfrowania), to jedyny publicznie dostępny algorytm zatwierdzony przez NSA do szyfrowania informacji poufnych. Skupię się na przedstawieniu jego wykorzystania w roli głównego szyfru blokowego. AES to obecnie podstawowy szyfr blokowy. Działa on na 16 bajtach naraz i oferuje trzy możliwe długości klucza: 16, 24 lub 32 bajty. Szereg trybów kodowania bloków stanowi część specyfikacji AES. W tym punkcie przyjrzymy się trybom ECB i CBC.

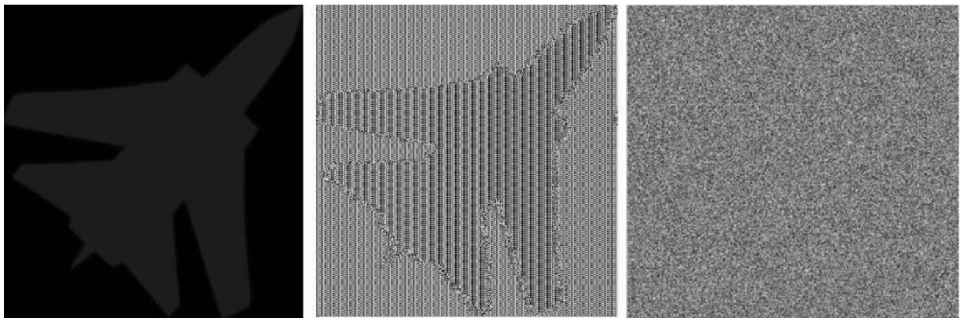
Poniższe przykłady kodu wykorzystują bibliotekę PyCryptodome, która została omówiona w punkcie „Advanced Encryption Standard (AES)" w rozdziale 5. Możesz ją zainstalować za pomocą narzędzia *pip*: `pip install pycryptodome`. Biblioteka umożliwia zastosowanie następujących trybów kodowania:

- `MODE_ECB` — tryb elektronicznej książki kodowej (ECB).
- `MODE_CBC` — tryb wiązania bloków zaszyfrowanych (CBC).
- `MODE_CFB` — tryb sprzężenia zwrotnego szyfrogramu (CFB).
- `MODE_PGP` — tryb PGP (ang. *Pretty Good Privacy* — całkiem niezła prywatność).

- `MODE_OFB` — tryb sprzężenia zwrotnego wyjścia (OFB).
- `MODE_CTR`— tryb licznikowy (CTR).
- `MODE_OPENPGP`— tryb OPENPGP (ang. *Open Pretty Good Privacy*).

Każdy blok składa się z 16 bajtów danych, a każdy tryb obsługuje klucze o długości 16, 24 i 32 bajtów. Szyfrowanie w większości trybów przebiega identyczne jak w CBC. W przypadku starszych wersji biblioteki domyślnie wykorzystywany jest tryb elektronicznej książki kodowej (ECB). Obecnie biblioteka wymaga jawnego określenia trybu kodowania. Tak czy inaczej, warto jawnie określić tryb, aby uniknąć niejasności co do jego wyboru.

Najprostszym trybem jest ECB. W trybie tym algorytm przetwarza osobno każdy 128-bitowy blok danych. Każdy blok jest następnie niezależnie szyfrowany za pomocą algorytmu AES z tym samym kluczem. Proces odszyfrowywania przebiega odwrotnie. Podczas pracy w trybie ECB identyczne bloki tekstu jawnego zostaną zaszyfrowane w ten sam sposób i dadzą identyczne bloki szyfrogramu lub zaszyfrowanego obrazu. Chociaż słabe strony tego trybu szyfrowania mogą nie być początkowo widoczne, to w przypadku szyfrowania obrazów od razu widać poważną lukę w zabezpieczeniach. Rysunek 6.5 pokazuje różnicę pomiędzy trybem ECB i innym dostępnym trybem.



Rysunek 6.5. Tryby kodowania w szyfrowaniu obrazu (po lewej oryginalny obraz, w środku tryb ECB, po prawej inny tryb)

Prosty przykład użycia trybu ECB

W pierwszym przykładzie przeanalizujemy szyfrowanie obrazu w trybie ECB. Rozpoczniemy od bitmapy, która zawiera wyraźny wzór (zobacz rysunek 6.6).

Rysunek 6.6. Samolot, rysunek w formacie BMP

Pierwszy krok to zaimportowanie biblioteki i ustalenie klucza. Chociaż obsługiwane są również klucze 24- i 32-bajtowe, to w tym przykładzie zastosuję klucz o długości 16 bajtów:

```
>>> from Crypto.Cipher import AES
>>> key = b"aaaabbbbccccdddd"
>>> cipher = AES.new(key, AES.MODE_ECB)
```

Następny krok to wczytanie pliku z obrazem (*samolot.bmp*). Binarna zawartość pliku zostanie zapisana w zmiennej `byteblock`:

```
>>> with open("samolot.bmp", "rb") as f:
>>> byteblock = f.read()
```

Rozmiar danych zapisanych w zmiennej `byteblock` musi być wielokrotnością 16 bajtów, inaczej pojawi się błąd: *Input strings must be a multiple of 16 in length* (pol. łańcuchy wejściowe muszą mieć rozmiar, który jest wielokrotnością 16). Długość `byteblock` będzie zależeć od rozmiaru obrazu. Do zbadania długości zmiennej możesz wykorzystać następujący kod:

```
>>> print(len(byteblock))
261654
```

Ponieważ długość łańcucha wejściowego do funkcji szyfrującej musi być wielokrotnością 16, sprawdzam, ile bajtów pozostanie w zmiennej, gdy pobiorę dane modulo 16. W tym przypadku będzie to 6 bajtów:

```
>>> print(len(byteblock)%16)
6
```

Moim celem jest teraz przeniesienie ciągu bajtów, którego długość jest wielokrotnością 16, do zmiennej, która pozwoli wyizolować z danych blok pozbawiony nadmiaru. W tym przykładzie zmienną tą będzie `byteblock_trimmed`:

```
byteblock_trimmed = byteblock[64:-6]
>>> print(len(byteblock_trimmed))
261584
>>> print(len(byteblock_trimmed)%16)
0
```

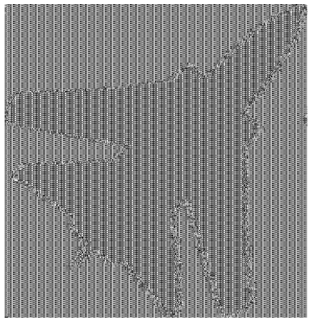
Aby mieć pewność, że obraz uda się zaszyfrować i odszyfrować bez utraty danych, muszę połączyć dwa zestawy bloków:

```
# W zmiennej byteblock_trimmed nie mogą się znajdować nadmiarowe bajty, inaczej wystąpi błąd
ciphertext = cipher.encrypt(byteblock_trimmed)
ciphertext = byteblock[0:64] + ciphertext + byteblock[-6:]
```

Teraz wystarczy tylko zapisać obraz ze zmiennej `ciphertext` do pliku:

```
with open("samolot_ecb.bmp", "wb") as f:
   f.write(ciphertext)
```

Wynikowy obraz pokazałem na rysunku 6.7.



Rysunek 6.7. Samolot zaszyfrowany w trybie ECB

Czas na odwrócenie całego procesu. Proces deszyfrowania będzie wyglądał następująco:

```
with open("samolot_ecb.bmp", "rb") as f:
     byteblock = f.read()
byteblock_trimmed = byteblock[64:-6]

plaintext = cipher.decrypt(byteblock_trimmed)
plaintext = byteblock[0:64] + plaintext + byteblock[-6:]

with open("d_samolot_ecb.bmp", "wb") as f:
     byteblock = f.write(plaintext)
```

Kod utworzy odszyfrowaną wersję obrazka, która nie rożni się niczym od oryginału (zobacz rysunek 6.8).

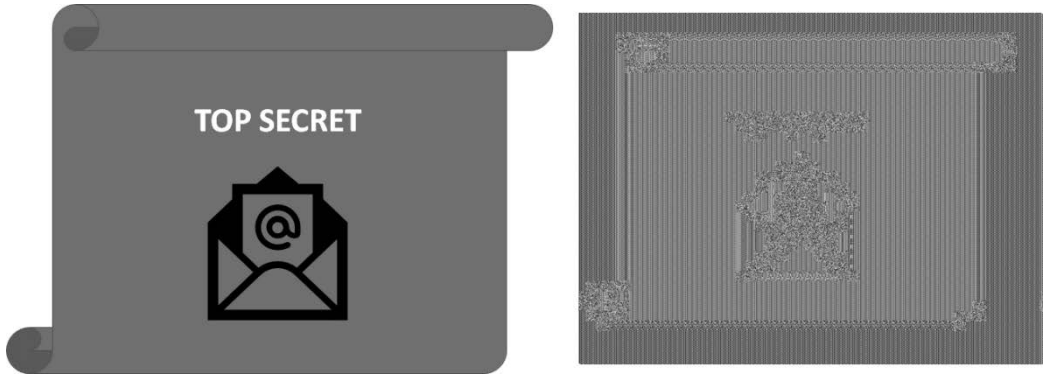
Rysunek 6.8. Rezultat odszyfrowywania obrazka, który zaszyfrowano w trybie ECB

W zależności od obrazu liczba bajtów, które nie tworzą pełnego bloku, będzie różna. Pokazany poniżej kod przechowuje tę liczbę w zmiennej `pad`. Dla uproszczenia kodu wartość tę pomnożyłem przez –1, tak aby uzyskać liczbę ujemną. Kod łączy wszystko, co już umiesz, i szyfruje obraz w formacie *.bmp*:

```
from Crypto.Cipher import AES
key = b"aaaabbbbccccdddd"
cipher = AES.new(key, AES.MODE_ECB)
# Szyfrowanie w trybie ECB
with open("./r06_sekretny_obraz.bmp", "rb") as f:
```

```
        byteblock = f.read()

pad = len(byteblock)%16 * -1
byteblock_trimmed = byteblock[64:pad]
ciphertext = cipher.encrypt(byteblock_trimmed)
ciphertext = byteblock[0:64] + ciphertext + byteblock[pad:]

with open("./e_r06_sekretny_obraz.bmp", "wb") as f:
    f.write(ciphertext)

# Odszyfrowywanie
with open("./e_r06_sekretny_obraz.bmp", "rb") as f:
byteblock = f.read()

pad = len(byteblock)%16 * -1
byteblock_trimmed = byteblock[64:pad]
plaintext = cipher.decrypt(byteblock_trimmed)
plaintext = byteblock[0:64] + plaintext + byteblock[pad:]

with open("./d_r06_sekretny_obraz.bmp", "wb") as f:
    byteblock = f.write(plaintext)

print("Gotowe")
```

Jednym z istotnych wniosków, które warto zapamiętać, jest to, że mapy bitowe zawierające duże jednolite obszary nie zostaną zaszyfrowane zgodnie z Twoimi oczekiwaniami. Istnieje szansa, że po zaszyfrowaniu mapy będą nadal zawierały informacje, które chcesz ukryć. Na rysunku 6.7 widać, że kontur samolotu pozostaje widoczny. Następny przykład trochę lepiej radzi sobie z zaciemnianiem obrazu, ale rezultat wciąż zawiera zbyt wiele informacji. Rezultat użycia powyższego kodu pokazałem na rysunku 6.9. Ta „luka w zabezpieczeniach" jest unikalna dla szyfrowania w trybie ECB i występuje tylko wtedy, gdy obraz jest plikiem, w którym informacje o kolejnych pikselach zapisane są w przewidywalnej kolejności, np. plikiem *.bmp*. Niektóre typy obrazów, takie jak pliki *.jpg*, nie są na to narażone.



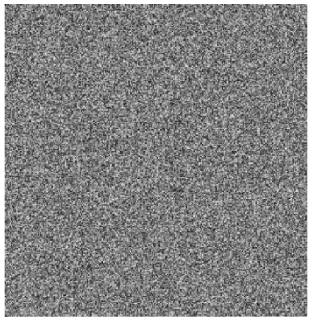
Rysunek 6.9. Wada szyfrowania w trybie ECB

Prosty przykład szyfrowania w trybie CBC

Wiesz już, jak szyfrować i odszyfrowywać w trybie ECB, zajmiemy się więc trybem CBC. Istotną różnicą pomiędzy tymi dwoma trybami jest użycie wektora inicjującego (IV):

```
from Crypto.Cipher import AES
iv = "1111222233334444".encode('UTF-8')
key = "aaaabbbbccccdddd".encode('UTF-8')

cipher = AES.new(key, AES.MODE_CBC, iv)
# Szyfrowanie w trybie CBC
with open("./samolot.bmp", "rb") as f:
    byteblock = f.read()

pad = len(byteblock)%16 * -1
byteblock_trimmed = byteblock[64:pad]
ciphertext = cipher.encrypt(byteblock_trimmed)
ciphertext = byteblock[0:64] + ciphertext + byteblock[pad:]

with open("./samolot_cbc.bmp", "wb") as f:
    f.write(ciphertext)

# Odszyfrowywanie
cipher = AES.new(key, AES.MODE_CBC, iv)

with open("./samolot_cbc.bmp", "rb") as f:
    byteblock = f.read()

pad = len(byteblock)%16 * -1
byteblock_trimmed = byteblock[64:pad]
plaintext = cipher.decrypt(byteblock_trimmed)
plaintext = byteblock[0:64] + plaintext + byteblock[pad:]

with open("./d_samolot_cbc.bmp", "wb") as f:
    byteblock = f.write(plaintext)
print("Gotowe")
```

Uruchomienie kodu da w wyniku zaszyfrowany obraz (rysunek 6.10), który nie zawiera luki znanej z trybu ECB.



Rysunek 6.10. Samolot zaszyfrowany w trybie CBC

Wykorzystanie wiedzy w praktyce

Posiadasz już wszystkie narzędzia potrzebne do stworzenia rozwiązania, które umożliwia szyfrowanie w wielu trybach kodowania AES. Poniższy kod pozwala określić nazwę pliku, klucz i wektor inicjujący. Kod ten możesz wykorzystać do przetestowania różnych trybów kodowania:

```
from Crypto.Cipher import AES

def Open_File(filename):
    with open(filename, "rb") as f:
        byteblock = f.read()
    return byteblock

def Save_File(filename, block):
    with open(filename,"wb") as f:
        f.write(block)

def Get_Padding(block):
l = len(block) %16
return (l * -1)

def Encrypt(cipher,read_filename, save_filename):
    block = Open_File(read_filename)
    pad = Get_Padding(block)
    block_trimmed = block[64:pad]
    ciphertext = cipher.encrypt(block_trimmed)
    ciphertext = block[0:64] + ciphertext + block[pad:]
    Save_File(save_filename, ciphertext)

def Decrypt(cipher,read_filename, save_filename):
    block = Open_File(read_filename)
    pad = Get_Padding(block)
    block_trimmed = block[64:pad]
    ciphertext = cipher.decrypt(block_trimmed)
    ciphertext = block[0:64] + ciphertext + block[pad:]
    Save_File(save_filename, ciphertext)

def Init_Cipher(key, mode, iv):
    cipher = AES.new(key, mode, iv)
return cipher

# Ustalenie klucza i wektora inicjującego
key = "aaaabbbbccccdddd".encode('UTF-8')
iv = "1111222233334444".encode('UTF-8')

# Dostępne w AES tryby kodowania bloków
# AES.MODE_ECB = 1
# AES.MODE_CBC = 2
# AES.MODE_CFB = 3
# AES.MODE_OFB = 5
# AES.MODE_CTR = 6
# AES.MODE_OPENPGP = 7


mode = AES.MODE_CBC

c = Init_Cipher(key,mode, iv)
```

```
Encrypt(c, "samolot.bmp", "e_samolot.bmp")

c = Init_Cipher(key,mode, iv)
Decrypt(c, "e_samolot.bmp", "d_samolot.bmp")
```

Steganografia

Steganografia (z gr. ukryte pisanie) to sztuka ukrywania danych w innym pliku, obrazie, filmie lub wiadomości. Termin ten został po raz pierwszy użyty w 1499 roku przez Jana Trithemiusa w książce *Steganographia*. Chociaż książka w rzeczywistości dotyczyła kryptografii i steganografii, tematyka ta została ukryta pod szyldem publikacji o czarach. W książce ukryto zaszyfrowane wiadomości. Po prawie 500 latach zamieszczone w niej kryptogramy zostały odnalezione i odszyfrowane. W rezultacie *Steganographia* nie jest już uważana za jeden z wczesnych nowożytnych traktatów demonologicznych, lecz za pierwszy europejski przykład ukrycia tajnych informacji w książce

W steganografii wiadomości często ukrywa się w obrazach, artykułach, listach lub innych tekstach. Tajną wiadomość można na przykład zapisać pomiędzy wierszami za pomocą niewidzialnego atramentu. W steganografii istotny nie jest sam artykuł, lecz ukryta w nim treść. W przypadku nieszyfrowanych wiadomości steganografia realizuje koncepcję bezpieczeństwa przez niejawność (ang. *security through obscurity*). Podstawowa różnica między steganografią i kryptografią polega na tym, że kryptografia koncentruje się na ochronie treści wiadomości, steganografia zaś zajmuje się ukrywaniem zarówno treści, jak i samej wiadomości.

Dzisiejsza steganografia skupia się na ukrywaniu informacji lub danych w plikach komputerowych lub w komunikacji elektronicznej. Z tego podrozdziału dowiesz się, jak z pomocą Pythona ukryć dane w różnych środkach elektronicznych. Wcześniej wspomniałem o module *cryptosteganography*. Ukrywanie danych rozpoczniemy właśnie od tej biblioteki, następnie przeanalizujemy też inne rozwiązania. Niektóre z ograniczeń tego modułu to zapis danych wyjściowych tylko w formacie *.png* i brak obsługi przypadków, w których ukrywany plik jest większy niż oryginalne wejście. Do przechowywania dużych bloków danych mogą więc być potrzebne większe obrazy.

Przechowywanie wiadomości w obrazie

Zajmiemy się teraz przechowywaniem danych w obrazie cyfrowym. Każdy obraz, niezależnie od formatu, składa się z wartości liczbowych zwanych pikselami. Piksele są odpowiednikiem komórek w ludzkim ciele i stanowią najmniejszy element konstrukcyjny obrazu. Każda wartość piksela reprezentuje kolor w danym punkcie. Kiedy przyjrzysz się obrazowi na poziomie pikseli, zauważysz, że składa się on z szeregu pikseli ułożonych w wierszach i kolumnach. Obrazy o większej liczbie pikseli zapewniają dokładniejsze odwzorowanie prezentowanych treści. W obrazach cyfrowych kolor jest reprezentowany przez trzy lub cztery składowe, takie jak cyjan, magenta, żółty i czarny (ang. *cyan, magenta, yellow, black/key color* — CMYK) lub czerwony, zielony i niebieski (ang. *red, green, blue* — RGB). W modelu RGB szeroka gama kolorów wynika z połączenia odcieni czerwieni, zieleni i niebieskiego. Kolor w modelu RGB opisują trzy wartości (czerwony, zielony, niebieski), z których każda jest 8-bitową liczbą z zakresu od 0 do 255. W zapisie cyfrowym kolor w formacie RGB jest reprezentowany przez trzy liczby, na przykład jeden z odcieni zieleni to `[124, 196, 143]`. W internecie dostępne są narzędzia pozwalające sprawdzić odcień dowolnej kombinacji. Każda wartość

w formacie RGB jest reprezentowana przez kod binarny. Kod ten składa się z 8-bitowych cyfr binarnych. Najbardziej znaczący jest lewy bit. Liczbę128 reprezentuje 10000000, a liczbę 177 — 10110001. Po prawej stronie znajduje się najmniej znaczący bit. Zmiana skrajnie prawej wartości ma więc mniejszy wpływ na wartość końcową. Niewielkie zmiany w najmniej znaczącym fragmencie nie są widoczne gołym okiem. Będzie to istotne podczas ukrywania danych za pomocą metody najmniej znaczącego bitu (ang. *least significant bit* — LSB). W 8-bitowym obrazie RGB na każdy piksel składają się trzy 8-bitowe kanały. Każdy kolor ma swój własny 8-bitowy kanał, a odcień każdej składowej zapisany jest w osobnym bajcie. O takim obrazie mówi się, że ma on 24-bitową głębię kolorów. Zapis ten pozwala reprezentować 16,7 miliona kolorów. Metoda LSB zastępuje bity w obrazie, ale nie wszystkie te bity są potrzebne do prezentacji obrazu. Tę samą metodę możesz wykorzystać do ukrycia danych w plikach audio. Inną metodą ukrywania danych w obrazach jest dołączanie dodatkowych bajtów na końcu pliku, w taki sposób, że plik z obrazem zachowuje techniczną poprawność. Dodatkowe bajty można też ukryć w metadanych. W większości przypadków spowoduje to jednak zwiększenie rozmiaru pliku. Podczas dodawania bitów do pliku tajna wiadomość może pojawić się także w nagłówku, który zawiera informacje takie jak typ pliku, głębia kolorów i rozdzielczość obrazu. Ponieważ każdy format ma jasno zdefiniowany koniec pliku (ang. *end of file*), dane można również ukryć bez modyfikowania/uszkadzania obrazu po znaku końca pliku.

Naszym celem będzie ukrycie wiadomości w obrazie za pomocą modułu *cryptosteganography*. Zacznę od kodu, potem pokażę Ci kilka narzędzi, które są pomocne w odkrywaniu ukrytych w obrazach danych.

Pierwszą rzeczą, którą musisz zrobić, jest zaimportowanie biblioteki `CryptoSteganography`:

```
>>> from cryptosteganography import CryptoSteganography
```

Następny krok to utworzenie klucza i przekazanie go do obiektu `CryptoSteganography`. Klucz jest niezbędny do odczytania ukrytej wiadomości. Kluczem może być wygenerowane lub dowolnie wybrane hasło:

```
>>> key = "1111222233334444!"
>>> crypto_steganography = CryptoSteganography(key)
```

Teraz utworzę trzy zmienne: pierwsza będzie zawierała nazwę oryginalnego obrazu, druga nazwę zmienionego lub zmodyfikowanego pliku, a trzecia wiadomość, którą chcesz ukryć w obrazie. Do ukrycia wiadomości posłuży mi metoda `hide()` obiektu `crypto_steganography`:

```
>>> origfile = "r06_sekretny_obraz.jpg"
>>> modfile = "steg_r06_sekretny_obraz.png"
>>> message = "Tajna wiadomość."
>>> crypto_steganography.hide(origfile, modfile, message)
```

Metoda `retrieve()` pozwoli wyodrębnić wiadomość z właśnie utworzonego obrazu. Gdy odzyskasz tekst, możesz wyświetlić go na ekranie i sprawdzić, czy nie nastąpiły w nim jakieś zmiany:

```
>>> secret = crypto_steganography.retrieve(modfile)
>>> print(secret)
Tajna wiadomość.
```

Przeanalizujmy, jak moduł zachowuje się w przypadku nieprawidłowego klucza. Zmień wartość zmiennej `key` na jakąś inną, a następnie przekaż nowy klucz do klasy `CryptoSteganography`:

```
>>> key = "InnyKlucz"
>>> crypto_steganography = CryptoSteganography(key)
```

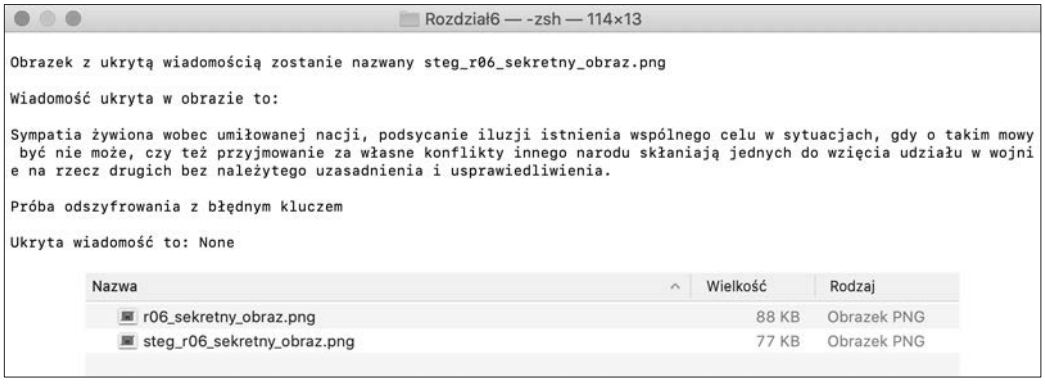
Spróbuj teraz wyodrębnić wiadomość za pomocą metody `retrieve()`. Tym razem po wyświetleniu wiadomości zobaczysz wartość `None`, która wskazuje, że zastosowano nieprawidłowy klucz:

```
>>> secret = crypto_steganography.retrieve(modfile)
>>> print(secret)
None
```

Skoro znasz już wszystkie komponenty, połączę je i pokażę Ci, w jaki sposób ukrycie informacji za pomocą steganografii zmienia oryginalny plik:

```
from cryptosteganography import CryptoSteganography

key = "1111222233334444!"
crypto_steganography = CryptoSteganography(key)

print()
print('Program szuka pliku r06_sekretny_obraz.png\n')
origfile = "./steg/r06_sekretny_obraz.png"
print('Obrazek z ukrytą wiadomością zostanie nazwany steg_r06_sekretny_obraz.png\n')
modfile = "./steg/steg_r06_sekretny_obraz.png"

secretMsg = ""
# Fragment „Pożegnalnej mowy Jerzego Waszyngtona, którą wygłosił, opuszczając stanowisko Prezydenta Stanów
# Zjednoczonych Ameryki”,
# https://www.pafere.org/2020/12/15/artykuly/pozegnalna-mowa-jerzego-waszyngtona/.
message1 = "Sympatia żywiona wobec umiłowanej nacji, podsycanie iluzji istnienia wspólnego celu "
message2 = "w sytuacjach, gdy o takim mowy być nie może, czy też przyjmowanie za własne konflikty "
message3 = "innego narodu skłaniają jednych do wzięcia udziału w wojnie na rzecz drugich "
message4 = "bez należytego uzasadnienia i usprawiedliwienia."
secretMsg = secretMsg.join([message1, message2, message3, message4])

crypto_steganography.hide(origfile, modfile, secretMsg)
secret = crypto_steganography.retrieve(modfile)
print("Wiadomość ukryta w obrazie to:\n")
print(secret)
print()

print('Próba odszyfrowania z błędnym kluczem\n')
key = "InnyKlucz"
crypto_steganography = CryptoSteganography(key)
secret = crypto_steganography.retrieve(modfile)
print('Ukryta wiadomość to: {} \n'.format(secret))
```

Kod spowoduje ukrycie cytatu Jerzego Waszyngtona w obrazku o nazwie *r06_sekretny_obraz.png*. Przekonwertowałem oryginalny obraz na format *.png*, aby Ci pokazać, jak ukrycie wiadomości wpłynie na rozmiar pliku. Oryginalny plik w tym przypadku nie musi być w formacie *.png*, ale ułatwia to porównanie oryginału z wersją zmodyfikowaną. Często podczas ukrywania danych w obrazie rozmiar pliku się zwiększa. W tym przypadku jednak (zobacz rysunek 6.11) oryginalny obraz miał 88 kB, a obraz z ukrytą wiadomością ma tylko 78 kB. Sprawdzając rozmiary plików, nie jesteśmy więc w stanie ustalić, który z obrazów zawiera ukrytą wiadomość.



Rysunek 6.11. Przykład steganografii

Zauważ, że bez odpowiedniego klucza ukryta wiadomość nie może zostać ujawniona. Zatem pojawia się pytanie: jak sprawdzić, czy w obrazie znajdują się ukryte dane? W internecie dostępnych jest wiele bezpłatnych narzędzi, za pomocą których można osadzać i wyodrębniać dane z obrazów. Jednym z narzędzi, które wykorzystałem kilka razy, jest Invisible Secrets. Przeanalizujemy teraz bardziej złożony przypadek z ukrywaniem danych binarnych.

Ukrywanie pliku binarnego w obrazie

W następnym przykładzie ukryję w obrazie plik multimedialny. Plik, który wykorzystamy, został pobrany z serwisu *file-examples.com*. Plik ma 764 kB. Jeśli chcesz zastosować ten sam plik, to znajdziesz go pod adresem *https://file-examples.com/index.php/sample-audio-files/sample-mp3-download/*.

W tym przykładzie moim celem jest wykorzystanie modułu `CryptoSteganography` do ukrycia pliku w obrazku. Im większy plik multimedialny, tym dłużej potrwa jego zaszyfrowanie i odszyfrowanie. Ponieważ wybrany przeze mnie plik audio jest za duży dla poprzedniego obrazka, w tym przykładzie zastosuję znacznie większe zdjęcie. Podczas pracy z tym przykładem spróbuj ukryć w obrazie więcej danych, niż pozwala na to jego rozmiar. Próba taka powinna zakończyć się powstaniem wyjątku *The message you want to hide is too long* („Wiadomość, którą chcesz ukryć, jest za długa"). Zacznę od omówienia poszczególnych składników kodu. Pierwszą rzeczą, którą musisz zrobić, jest zaimportowanie biblioteki `CryptoSteganography`:

```
>>> from cryptosteganography import CryptoSteganography
```

Następny krok to otwarcie pliku i zapisanie danych binarnych w zmiennej, którą w tym przypadku będzie `message`:

```
>>> mediafile = "file_example_MP3_700KB.mp3"
>>> message = None
>>> with open(mediafile, "rb") as f:
>>>     message = f.read()
```

Potem tworzę klucz i przekazuję go do obiektu `CryptoSteganography`. Klucz jest niezbędny do odzyskania ukrytej wiadomości. Kluczem może być wygenerowane lub dowolnie wybrane hasło:

```
>>> key = "1111222233334444!"
>>> crypto_steganography = CryptoSteganography(key)
```

Mogę teraz ukryć plik w obrazie. Przypiszę nazwy plików do zmiennych:

```
>>> origfile = "psy.jpg"
>>> modfile = "steg_audio_psy.png"
>>> crypto_steganography.hide(origfile, modfile, message)
```

Teraz, gdy ukryłem już plik audio, przyszedł czas, aby go wyodrębnić. Możesz skorzystać z poprzedniego obiektu `crypto_steganography`. W poniższym kodzie tworzę jednak jego nową instancję, która pozwoli Ci zmienić lub zmodyfikować klucz. W tym przykładzie zastosuję ten sam klucz. Ponadto ustalam nazwę obrazka z zaszyfrowanym plikiem i nazwę pliku, w którym zapiszę odszyfrowaną muzykę:

```
>>> key = "1111222233334444!"
>>> crypto_steganography = CryptoSteganography(key)
>>> modfile = 'steg_audio_psy.png'
>>> decrypted = 'odszyfrowana_probka.mp3'
```

Metoda `retrieve()` pozwoli wyodrębnić z pliku dane audio i zapisać je pod nazwą określoną w zmiennej `decrypted`:

```
>>> secret_bin = crypto_steganography.retrieve(modfile)
>>> with open(decrypted, 'wb') as f:
>>>     f.write(secret_bin)
```

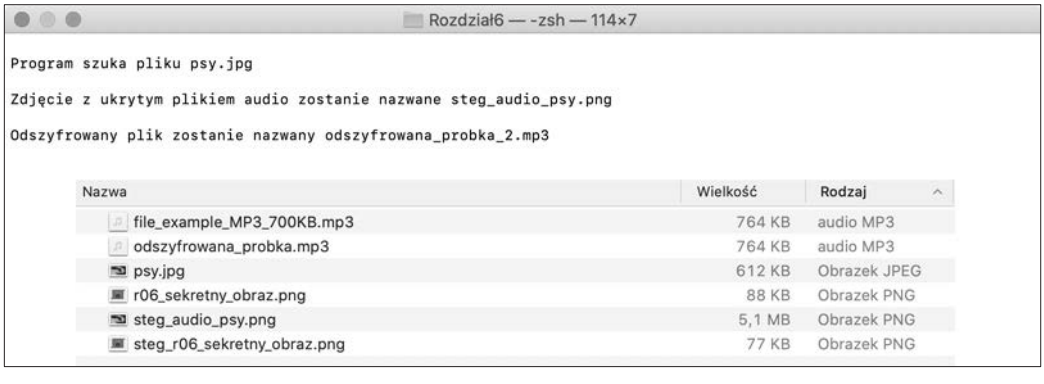
Połączę wszystko w działający kod, który ukryje plik multimedialny w zdjęciu moich psów o nazwie *psy.jpg*. W obrazie pokazanym na rysunku 6.12 nie ma nic specjalnego. Jest to po prostu zdjęcie moich dwóch psów zrobione telefonem komórkowym. Plik ten jest wystarczająco duży, aby można było ukryć w nim plik MP3 o rozmiarze 700 kB.

Rysunek 6.12. Zdjęcie moich psów w wysokiej rozdzielczości

Oto kod, który pobierze plik *.mp3* i ukryje go w dużym pliku *.jpg*:

```
from cryptosteganography import CryptoSteganography

# Otwieranie pliku audio
mediafile = './steg/file_example_MP3_700KB.mp3'
message = None

with open(mediafile, "rb") as f:
   message = f.read()

print()
print('Program szuka pliku psy.jpg\n')
origfile = "./steg/psy.jpg"
print('Zdjęcie z ukrytym plikiem audio zostanie nazwane steg_audio_psy.png\n')
modfile = "./steg/steg_audio_psy.png"

key = "1111222233334444!"
crypto_steganography = CryptoSteganography(key)
crypto_steganography.hide(origfile, modfile, message)

print('Odszyfrowany plik zostanie nazwany odszyfrowana_probka_2.mp3 \n')
decrypted = './steg/odszyfrowana_probka_2.mp3'
secret_bin = crypto_steganography.retrieve(modfile)

# Zapis danych w nowym pliku
with open(decrypted, 'wb') as f:
    f.write(secret_bin)
```

Rysunek 6.13 pokazuje wynik uruchomienia kodu. Kod pobiera plik multimedialny i ukrywa go w zdjęciu *psy.jpg*. Oryginalny rozmiar zdjęcia to 612 kB. Po osadzeniu pliku multimedialnego rozmiar wyjściowego pliku to 5,1 MB. Kod wyodrębnia również plik audio z obrazu i zapisuje go jako *odszyfrowana_probka_2.mp3*. Gdy porównasz dwa pliki *.mp3*, przekonasz się, że są one identyczne.



Rysunek 6.13. Steganografia, przykład z plikiem audio

Praca z dużymi obrazami

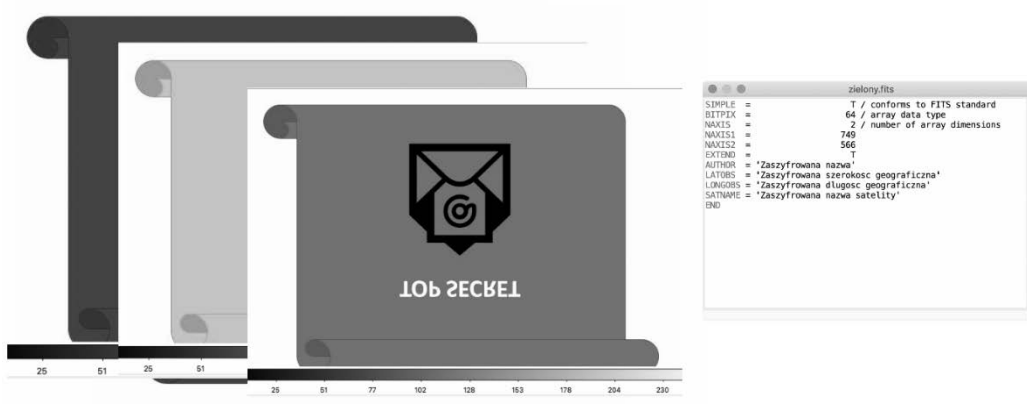
Skoro jesteśmy już w temacie obrazów i możliwości ukrywania w nich danych, chciałbym przedstawić Ci format FITS (ang. *Flexible Image Transport System* — elastyczny system transportu obrazu). Jest to otwarty standard, który definiuje format pozwalający na przesyłanie, przetwarzanie i przechowywania danych w jednym pliku. FITS jest zwykle wykorzystywany do przechowywania danych z satelitów lub danych z badań astronomicznych. Ilość danych w pliku może być astronomiczna (gra słów celowa). Dane w pliku można przedstawić w postaci kilku struktur. W części nagłówkowej pliku znajduje się czytelny dla człowieka tekst, który ułatwia interpretację danych. Nagłówek może zawierać parametry takie jak informacje o tym, gdzie zostało zrobione zdjęcie lub który satelita/teleskop je wykonał, oraz dowolną ilość metadanych stworzonych przez autora. Jedynym ograniczeniem związanym z metadanymi w nagłówku jest to, że etykieta może mieć maksymalnie osiem znaków. Plik FITS składa się z co najmniej jednej pary nagłówek – jednostka danych (para taka jest nazywana HDU — ang. *Header-Data Unit*). Pierwsza utworzona jednostka HDU nosi nazwę podstawowej jednostki HDU lub tablicy podstawowej (ang. *primary array*). Podstawowa jednostka HDU może być pusta lub zawierać *N*-wymiarową tablicę pikseli, na przykład jednowymiarowe widmo, dwuwymiarowy obraz lub trójwymiarową kostkę danych. Najlepszym sposobem na zrozumienie standardu FITS jest analiza kodu w Pythonie. Do uruchomienia poniższego kodu wymagana jest biblioteka *astropy*. Inną biblioteką, którą możesz zastosować, jest PyFITS. Moduł `.io` z FITS nie różni się od PyFITS niczym poza nazwą. Moduł ten został po prostu przeportowany do biblioteki *astropy*. Bibliotekę *astropy* można zainstalować za pomocą komendy `pip install astropy`. Utworzę plik FITS o nazwie *losowa_tablica.fits*, a następnie umieszczę w nim tablicę losowych wartości. Plik będzie zawierał domyślny nagłówek i dane.

```
import numpy as np
from astropy.io import fits

file_name = "./fits/losowa_tablica.fits"
hdu = fits.PrimaryHDU()
hdu.data = np.random.random((128,128))
# Zauważ, że dodanie danych powoduje powstanie automatycznie wygenerowanego nagłówka
hdu.writeto(file_name, overwrite=True)

data = fits.getdata(file_name)
header = fits.getheader(file_name)

print(header)
print()
print(data)
```

Skoro znasz już elementy, które składają się na plik FITS, możemy zająć się przypadkiem ukrywania części informacji z nagłówka przechowywanego obrazu. Wiesz już, jak zaszyfrować łańcuch znaków, więc zakładam, że zaszyfrowałeś dane swoim ulubionym algorytmem. Ponieważ długość etykiet jest ograniczona do ośmiu znaków, nie są one najlepszym kandydatem do szyfrowania. Ograniczenie to nie dotyczy jednak ich treści. Jeśli istnieją dane, które chcesz zachować w tajemnicy, ten oto kod Pythona załatwi sprawę:

```
import matplotlib.pyplot as plt
import numpy as np

from PIL import Image
from astropy.io import fits
from astropy.visualization import astropy_mpl_style

location_lat = "Zaszyfrowana szerokosc geograficzna"
location_long = "Zaszyfrowana dlugosc geograficzna"
author = "Zaszyfrowana nazwa"
satellite = "Zaszyfrowana nazwa satelity"

# Wczytanie obrazka do tablicy pikseli
img_file = Image.open('./fits/r06_sekretny_obraz.jpg')
xsize, ysize = img_file.size
print("Rozmiar obrazu: {} x {}".format(xsize, ysize))
plt.style.use(astropy_mpl_style)
plt.imshow(img_file)

r, g, b = img_file.split()
r_data = np.array(r.getdata())
g_data = np.array(g.getdata())
b_data = np.array(b.getdata())
print(r_data.shape)

r_data = r_data.reshape(ysize, xsize)
g_data = g_data.reshape(ysize, xsize)
b_data = b_data.reshape(ysize, xsize)

red = fits.PrimaryHDU(data=r_data)
red.header["AUTHOR"] = author
red.header["LATOBS"] = location_lat
red.header["LONGOBS"] = location_long
red.header["SATNAME"] = satellite
red.writeto('./fits/czerwony.fits', overwrite=True)

green = fits.PrimaryHDU(data=g_data)
green.header["AUTHOR"] = author
green.header["LATOBS"] = location_lat
green.header["LONGOBS"] = location_long
green.header["SATNAME"] = satellite
green.writeto('./fits/zielony.fits', overwrite=True)

blue = fits.PrimaryHDU(data=b_data)
blue.header["AUTHOR"] = author
blue.header["LATOBS"] = location_lat
blue.header["LONGOBS"] = location_long
blue.header["SATNAME"] = satellite
blue.writeto('./fits/niebieski.fits', overwrite=True)
```

Na podstawie sekretnego obrazka z początku rozdziału kod tworzy trzy unikalne pliki FITS zawierające dane z trzech kanałów: czerwonego, zielonego i niebieskiego. Na rysunku 6.14 możesz zauważyć, że w nagłówku pliku *zielony.fits* znajdują się „zaszyfrowane” informacje. Jak można się spodziewać, równie łatwo da się zaszyfrować sam obraz.



Rysunek 6.14. Pliki FITS

Podsumowanie

W tym rozdziale przedstawiłem kilka bibliotek wspierających proces szyfrowania obrazów i steganografię. Ważne jest, aby zrozumieć, jak działają różne tryby kodowania bloków i jakie mają one zastosowanie w przypadku plików graficznych. Rozdział zakończyło omówienie biblioteki *cryptosteganography*. Biblioteka ta służy do ukrywania różnych danych w obrazach. Ukrycie wiadomości w obrazie może wpłynąć na jego właściwości. Stało się to oczywiste, gdy ukryłem w zdjęciu 30-sekundowy plik *.mp3*. Proces odszyfrowywania wymaga znajomości tajnego klucza. Jeśli odbiorca nie zna klucza, to wiadomość pozostanie tajna, nawet gdy odbiorca odkryje, że wizualne właściwości pliku uległy zmianie. Na koniec przedstawiłem format plików FITS. Chociaż format ten na pierwszy rzut oka może nie wydawać się jakiś szczególny, to ilość danych, które można za jego pomocą przechowywać i szyfrować, jest niesamowita.

MASZ COŚ DO UKRYCIA?
ZASZYFRUJ TO W PYTHONIE!

Dzięki kryptografii możemy w dużym stopniu zabezpieczyć swoje dane. Z szyfrowaną komunikacją wiążą się jednak kontrowersje i sprzeczności interesów. Przestępcy, ale również rządy, policja i służby wywiadowcze dążą do uzyskania możliwości wglądu we wszystkie formy komunikacji. Świat toczy wojnę o to, co można zaszyfrować, co powinno być zaszyfrowane i kto powinien dysponować kluczem pozwalającym odczytać zaszyfrowane wiadomości należące do innej osoby. W tej sytuacji zrozumienie, czym jest szyfrowanie, jak je stosować i jak się upewniać co do autentyczności i poufności otrzymywanych danych, staje się niezwykle ważne.

Ta książka jest przystępnym wprowadzeniem do kryptografii i bibliotek kryptograficznych Pythona. Omówiono tu podstawowe koncepcje z tej dziedziny, najważniejsze algorytmy i niezbędny zakres podstaw matematycznych: liczby pierwsze, teorię grup czy generatory liczb pseudolosowych. Wyjaśniono, czym są poufność, autentyczność i integralność wiadomości. Zaprezentowano najciekawsze biblioteki kryptograficzne Pythona i dokładnie pokazano, w jaki sposób można je wykorzystywać we własnych implementacjach. Wiele z prezentowanych koncepcji, między innymi kryptografia klucza publicznego i implementacja kryptografii krzywych eliptycznych, zostało przedstawionych w praktyce, za pomocą kodu Pythona, tak aby można było wymieniać dane w bardzo bezpiecznym formacie przez niezabezpieczony kanał.

W KSIĄŻCE:

- podstawy Pythona i kryptografii
- protokoły kryptograficzne i matematyka kryptograficzna
- kryptoanaliza za pomocą kodu Pythona
- kryptografia wizualna: biblioteki, algorytmy, tryby kodowania
- integralność wiadomości
- tworzenie rozwiązań kryptograficznych w Pythonie

SHANNON W. BRAY zajmuje się IT od 1997 roku, wcześniej służył w marynarce wojennej USA. Od kilkunastu lat interesuje się kryptografią i cyberbezpieczeństwem, obecnie przygotowuje doktorat z tej dziedziny. Zdobył liczne certyfikaty branżowe, w tym CISM, Security+ i CASP+. Hobbystycznie pracuje nad projektami dla domu, takimi jak domowy system bezpieczeństwa oparty na Pythonie, kryptografii i dronach.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-7729-5
9 788328 377295
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 69,00 zł