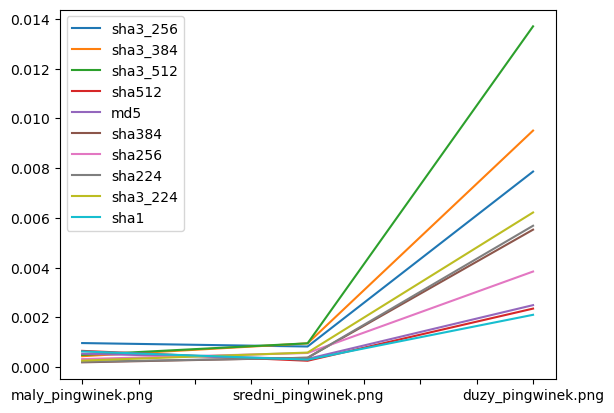
# Wybrane zagadnienia kryptograficzne – sprawozdanie z laboratoriów

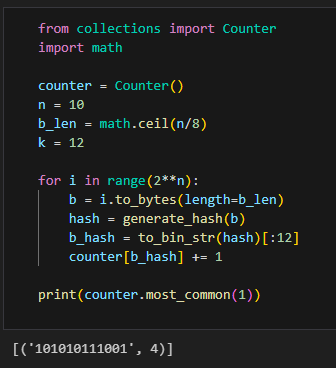
Jakub Grabowski (151825), L11

## Laboratorium 3 – funkcje skrótu

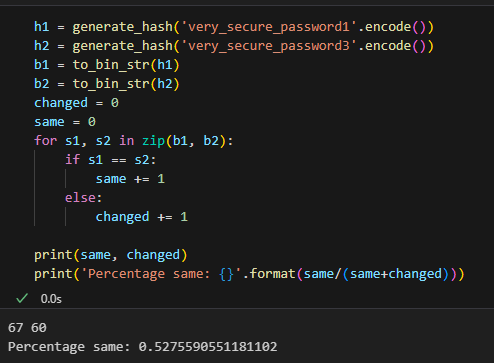
1. Aplikacja została zaimplementowana jako notatnik Jupyter i napisana w Pythonie. Aplikacja zawiera funkcję pomocniczą wybierającą metodę hashowania, skrypt badający szybkość hashowania trzech różniących się wielkością plików PNG przy pomocy różnych metod, skrypt badający ilość kolizji dla pierwszych 12 bitów skrótu dla 2^n kolejnych kombinacji bitów w bajtach (np. dla n=10 kombinacje pomiędzy bitami (hex): [00 00;04 00)) oraz skrypt badający, czy zmiana jednego bitu w słowie hashowanym zmienia bity wyniku z prawdopodobieństwem ok. p=0,5.
2. Sól jest używana do zwiększenia bezpieczeństwa gwarantowanego przez kombinację hasła i hashowania hasła. Jeżeli adwersarz przejąłby bazę danych z zahashowanymi hasłami, do których nie była dodana sól, to mógłby teoretycznie znaleźć dowolne słowo, którego rezultatem będzie ten hash i użyć go jako „hasła”. W momencie, w którym dodamy do hasła wpisywanego pewną losową sól przechowywaną w innej bazie danych możemy osłabić to działanie, gdyż adwersarz musiałby potencjalnie (zakładając, że zna i sól, i wynik hashowania) znaleźć takie słowo, które po doklejeniu do niego tej konkretnej soli da ten dokładny wynik hashowania (co jest rzędy wielkości trudniejsze).
3. Niestety, już w 2008 CMU SEI konkludowało w swoich badaniach, że ataki kolizyjne (analogiczne do tych opisanych w paragrafie wyżej) są jak najbardziej możliwe dla MD5. Jest to spowodowane faktem, że wielkość MD5 i jej skomplikowanie jest niewystarczające porównując je do możliwości dzisiejszych procesorów.
4. Wyniki:



Rysunek 1 - różnice w czasie hashowania dla plików różnej wielkości



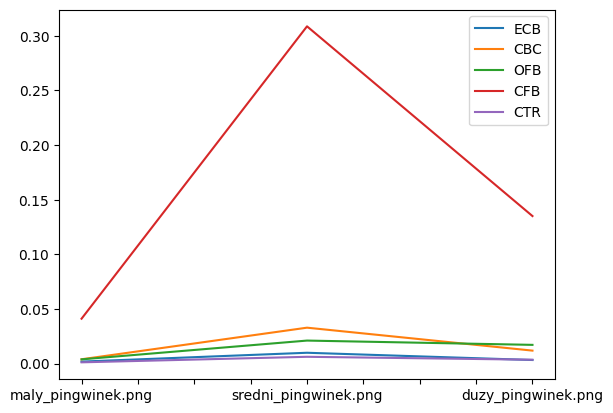
Rysunek 2 - widać najczęstszą kolizję na pierwszych 12b



Rysunek 3 - procent niezmienionych bitów w przybliżeniu równy założonemu 0,5

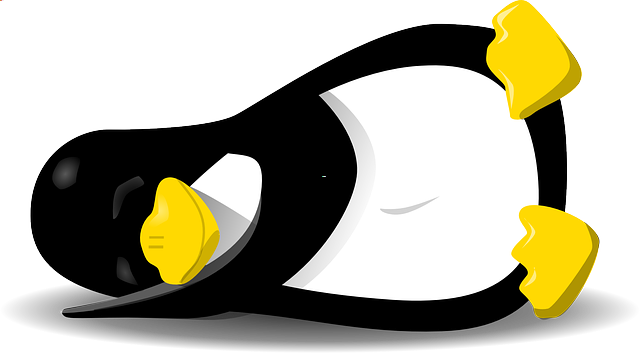
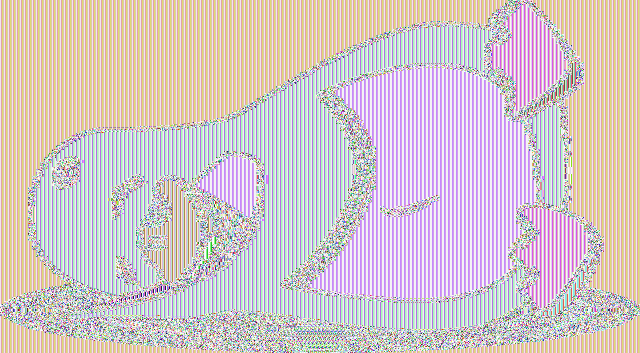
## Laboratorium 4 – Tryby pracy szyfrów blokowych

1. Wyniki badania czasów pracy AES w różnych trybach pokazują, że czas wykonywania dla plików o wielkości parudziesięciu kB do paru MB nie różni się znacznie, ponadto czas szyfrowania i deszyfrowania jest praktycznie taki sam. Większość trybów badanych ma bardzo zbliżone czasy wykonania: ECB i CTR działają najszybciej, CBC i OFB nieco wolniej, zaś tryb CFB jest wielokrotnie wolniejszy od wszystkich innych.



Rysunek 4 - wyniki szyfrowania plików różnymi trybami

1. Błędy choć jednego bitu na kluczu powodowały zupełnie niepoprawne deszyfrowanie w każdym z trybów. Należy tu zauważyć, że dla trybu ECB jest to najmniej widoczne, gdyż tryb ECB szyfruje pliki w taki sposób, że nadal widać w nim dobrze, co zostało zaszyfrowane – ta właściwość nie zanika przy deszyfrowaniu. Przy błędzie jednego bitu dla pliku zaszyfrowanego po odszyfrowaniu trybami ECB błędy pojawiły się lokalnie w obrębie bloku deszyfrowanego (pełna korupcja bloku). Dla CBC pojawiły się one w obrębie bloku deszyfrowanego (pełna korupcja) oraz następnego (zlokalizowany błąd w miejscu w bloku, gdzie nastąpił błąd oryginalny). Dla OFB i CTR błąd nastąpił tylko na bajcie uszkodzonym i nie został w ogóle rozpropagowany. Dla CFB propagacja błędu trwała od dokładnego bajtu uszkodzonego aż do następnego bajtu na tej pozycji w nowym bloku (ten bajt był już poprawny, dalszej propagacji nie było).



Rysunek 5 - pingwin odszyfrowany CBC z błędem w pliku - widać plamkę na skrzydle (błąd lokalny)

Rysunek 6- pingwin widoczny mimo zaszyfrowania ECB

ORG

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:7b:7c:1a:d0:93:d8:90:98:db:80:f3:6e:59:49:96:3b:BLOCK:2c:d7:31:b6:6e:68:79:72:5b:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

ECB

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:7d:6c:98:a4:15:c5:63:a4:a7:8d:0b:eb:0c:58:a3:37:BLOCK:2c:d7:31:b6:6e:68:79:72:5b:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

CBC

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:86:f8:8d:9f:96:f8:ae:1f:e4:b9:c2:e7:47:3c:3b:85:BLOCK:2c:d7:31:b6:6e:68:79:72:5a:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

OFB

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:7b:7c:1a:d0:93:d8:90:98:da:80:f3:6e:59:49:96:3b:BLOCK:2c:d7:31:b6:6e:68:79:72:5b:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

CFB

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:7b:7c:1a:d0:93:d8:90:98:da:92:db:89:8c:cf:bb:ef:BLOCK:40:f4:27:af:70:c5:84:20:4c:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

CTR

:5a:57:1e:af:33:32:c8:c9:b9:ed:44:f4:9c:c8:bc:01:BLOCK:3c:1c:1b:f9:58:45:6d:76:3e:3d:94:9a:5d:0b:6f:a1:BLOCK:fd:76:e9:2c:d2:58:ce:6b:51:61:c2:87:96:58:7d:cd:BLOCK:7b:7c:1a:d0:93:d8:90:98:da:80:f3:6e:59:49:96:3b:BLOCK:2c:d7:31:b6:6e:68:79:72:5b:83:f2:88:c9:c5:fd:91:BLOCK:63:4f:f6:24:d3:e2:c3:94:76:a1:93:7e:2a:40:ab:fd:BLOCK:c7:08:ca:81:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:0c:BLOCK

1. Implementacja CBC przy pomocy ECB:

def cbc\_encrypt(data, key, iv):

    cipher = AES.new(key=key, mode=AES.MODE\_ECB)

    c\_bytes = b''

    prev\_block = iv

    for i in range(0, len(data), AES.block\_size):

        new\_block = data[i : i + AES.block\_size]

        xor\_block = byte\_xor(new\_block, prev\_block)

        en\_block = cipher.encrypt(xor\_block)

        prev\_block = en\_block

        c\_bytes += en\_block

    return c\_bytes

def cbc\_decrypt(data, key, iv):

    cipher = AES.new(key=key, mode=AES.MODE\_ECB)

    c\_bytes = b''

    prev\_block = iv

    for i in range(0, len(data), AES.block\_size):

        new\_block = data[i : i + AES.block\_size]

        de\_block = cipher.decrypt(new\_block)

        xor\_block = byte\_xor(de\_block, prev\_block)

        c\_bytes += xor\_block

        prev\_block = new\_block

    return c\_bytes