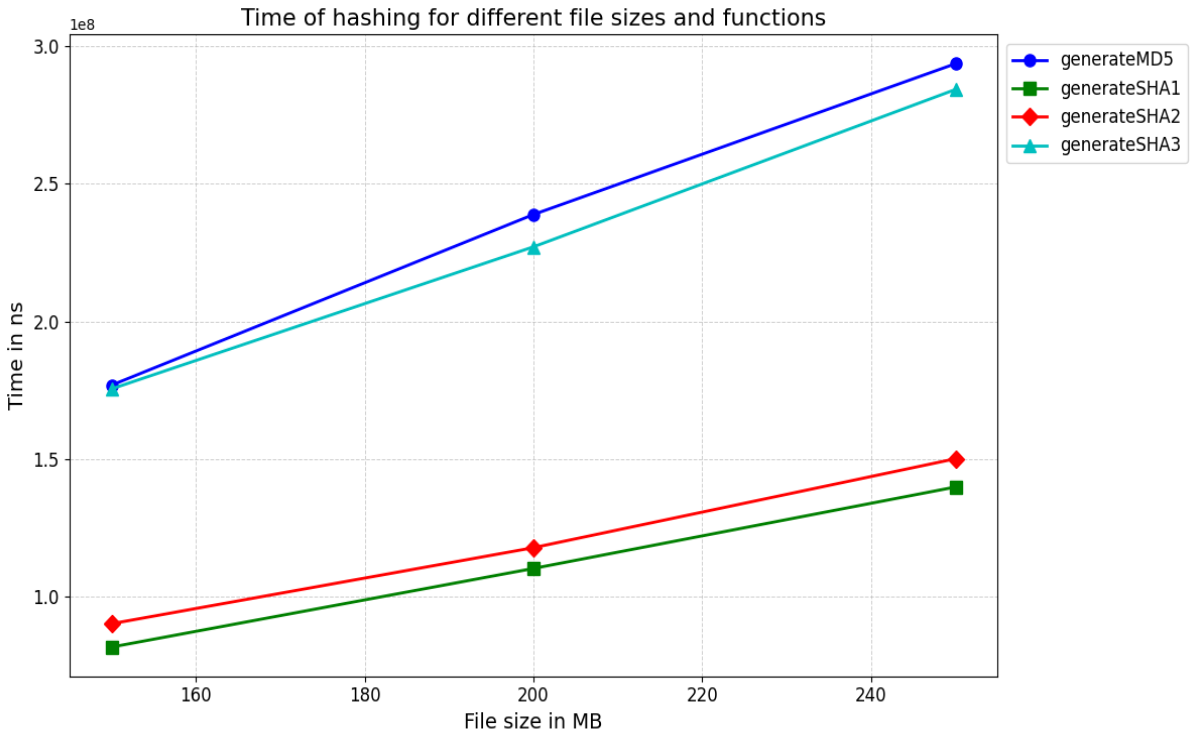
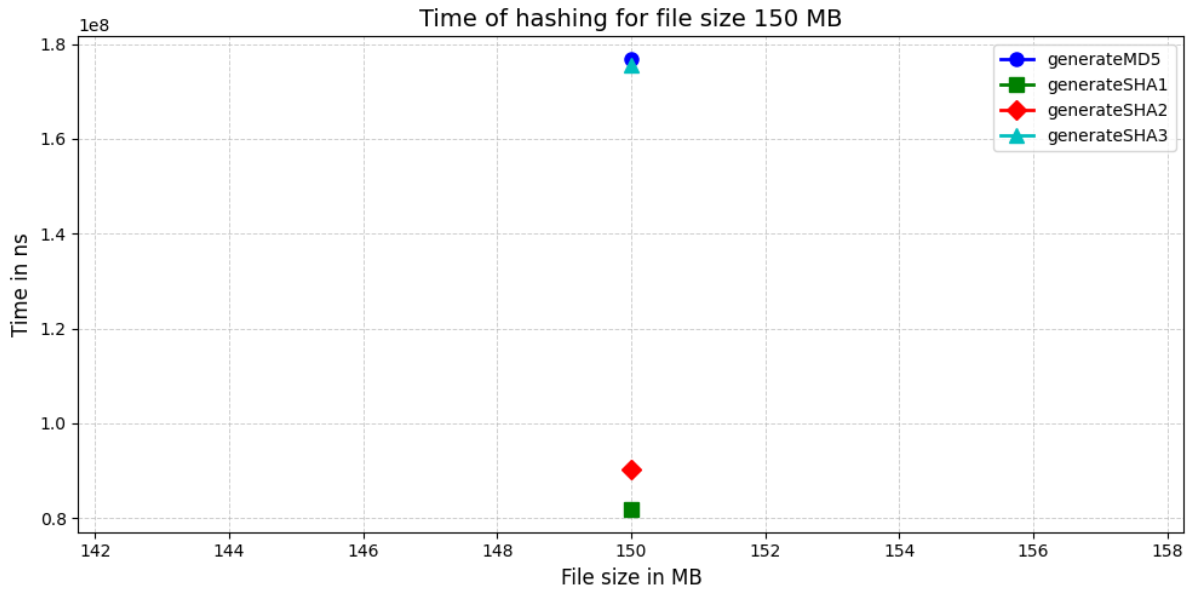
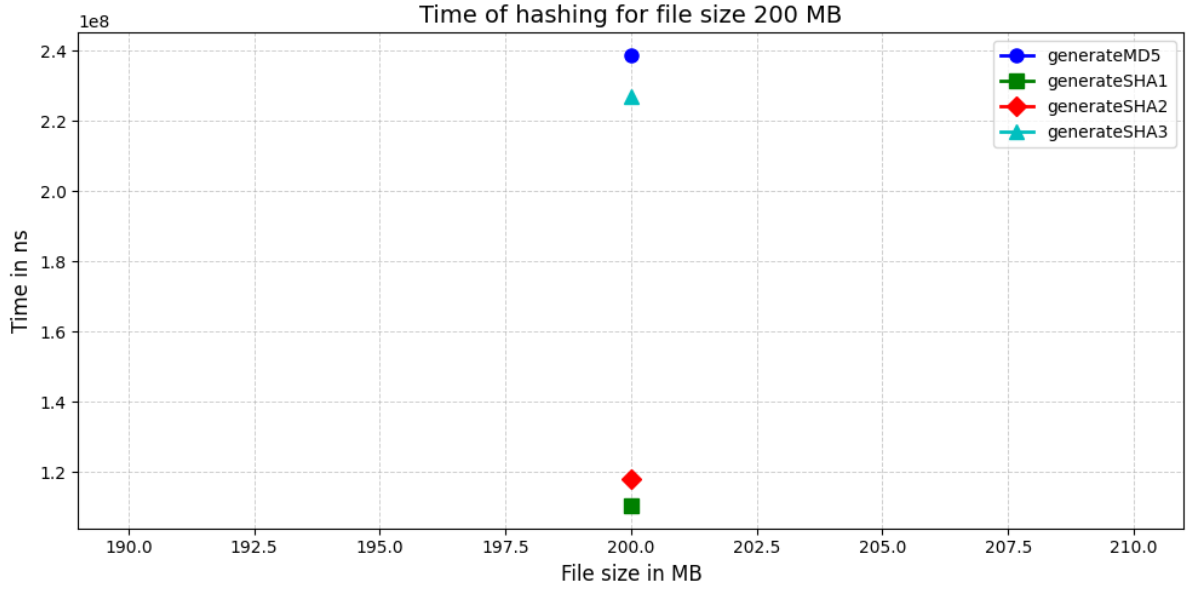
Miłosz Sobol 155905 Funkcje skrótu

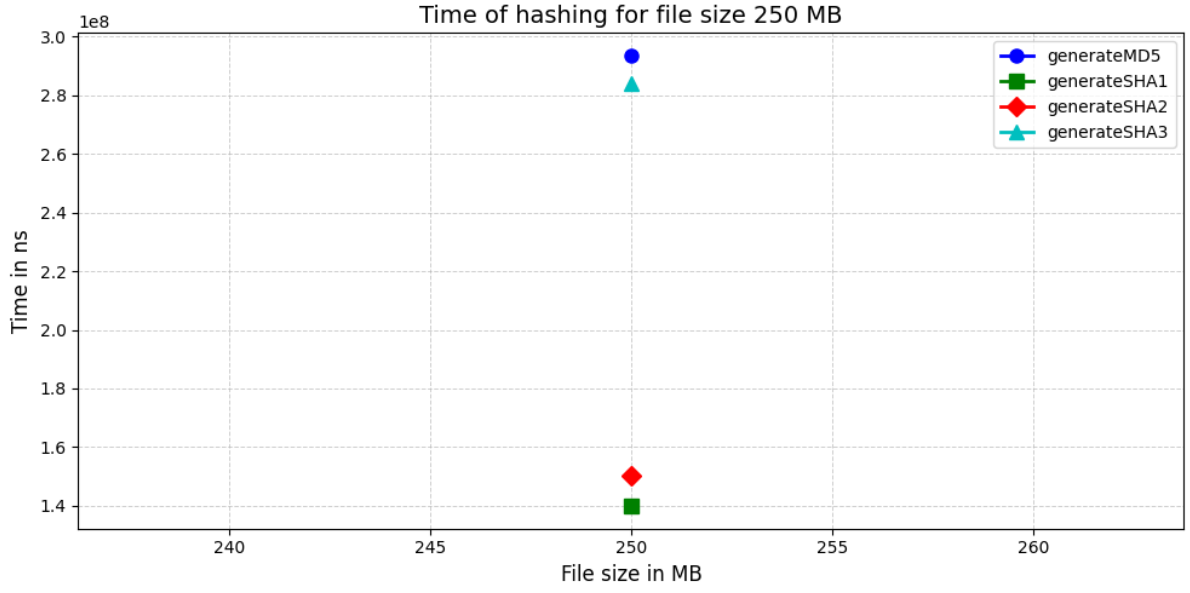
1. Zestawienie zależności czasu potrzebnego na wykonanie funkcji skrótu do wielkości przetwarzanego pliku.



Widoczny jest liniowy wzrost czasu zależnego od wielkości pliku przyjmowanego na wejście funkcji. Czasy te, mimo sporej wielkości plików, są bardzo niskie, świadczy to o niskiej złożoności obliczeniowej haszowania. Jest to zarówno plus jak i minus, niska złożoność obliczeniowa pozwala na stosunkowo szybkie przeprowadzanie ataków takich jak „atak urodzinowy” polegający na znalezieniu kolizji funkcji haszującej.

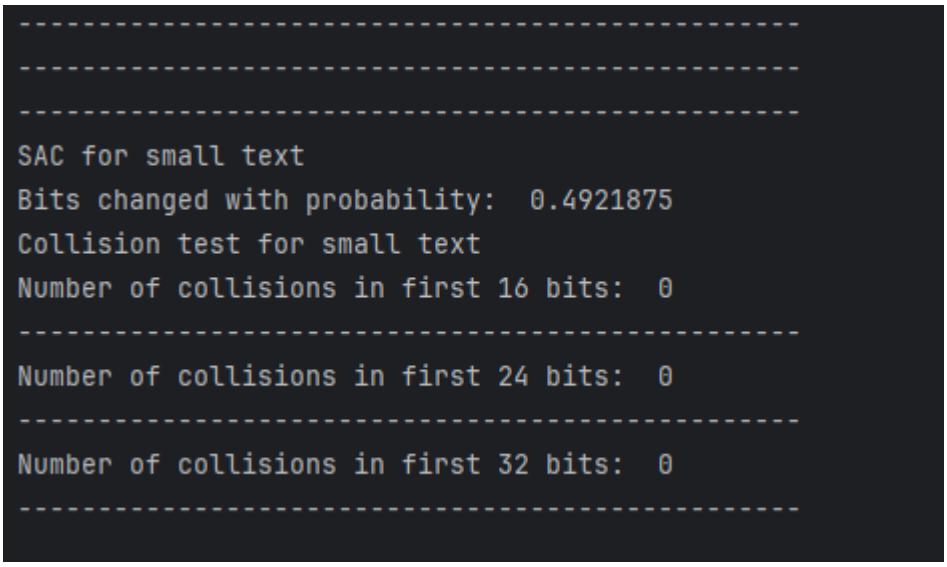

Time of hashing for file size 150 MB
1e8
1.8
1.6
1.4
1.2
1.0
0.8
Time in ns
142
144
146
148
150
152
154
156
158
File size in MB
generateMD5
generateSHA1
generateSHA2
generateSHA3


Time of hashing for file size 200 MB
1e8
2.4
2.2
2.0
1.8
1.6
1.4
1.2
Time in ns
190.0
192.5
195.0
197.5
200.0
202.5
205.0
207.5
210.0
File size in MB
generateMD5
generateSHA1
generateSHA2
generateSHA3


Time of hashing for file size 250 MB
1e8
3.0
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
Time in ns
240
245
250
255
260
File size in MB
generateMD5
generateSHA1
generateSHA2
generateSHA3

Na przedstawionych na poprzedniej stronie wykresach możemy zaobserwować pewną rozbieżność. Haszowanie za pomocą funkcji skrótu MD5 zajmuje niemalże tyle samo czasu co za pomocą SHA384. Owa rozbieżność najprawdopodobniej wynika z specyfikacji implementacji funkcji MD5 w Pythonowej bibliotece „hashlib” (https://stackoverflow.com/questions/59955854/what-is-md5-md5-and-why-is-hashlib-md5-so-much-slower).

2. Test SAC



Test został parokrotnie przeprowadzony dla plików o różnych rozmiarach (Na obrazku widoczny wynik dla pliku o wielkości 150MB oraz funkcji haszującej SHA384). Za każdym razem wynik oscylował w okolicach wartości 0.5.

3. Test kolizji

```
smallText = "ko"
#zad3
print("Collision test for small text")
res = solution.testCollision(smallText.encode(), numberOfTests: 1000 , solution.generateMD5Digest, numberOfTestingBytes: 2)
print('Number of collisions in first 16 bits: ', res)
printLine()
res =solution.testCollision(smallText.encode(), numberOfTests: 1000, solution.generateMD5Digest, numberOfTestingBytes: 3)
print('Number of collisions in first 24 bits: ', res)
printLine()
res =solution.testCollision(smallText.encode(), numberOfTests: 1000, solution.generateMD5Digest, numberOfTestingBytes: 4)
print('Number of collisions in first 32 bits: ', res)
printLine()
```

Mimo wykorzystania funkcji haszującej MD5, test charakteryzował się zupełnym brakiem kolizji dla stringa o długości 3 znaków. Poniżej widoczne wyniki dla kolejno, słowa „k” oraz słowa „ko”.

```
Collision test for small text
Number of collisions in first 16 bits:  16
------------------------------------------------
Number of collisions in first 24 bits:  16
------------------------------------------------
Number of collisions in first 32 bits:  11
------------------------------------------------
```

```
Collision test for small text
Number of collisions in first 16 bits:  0
------------------------------------------------
Number of collisions in first 24 bits:  1
------------------------------------------------
Number of collisions in first 32 bits:  0
------------------------------------------------
```

4. Implementacja

```
def generateAll(self,text: str, fileSize: str):
    res = dict()
    keys = self.keys
    functionNames = self.functionNames
    times = dict()


    for idx, functionName in
enumerate(functionNames):
        start = time.time_ns()
```

```
        res[keys[idx]]=
functionName(text.encode())
        end = time.time_ns()
        times[(keys[idx], fileSize,
functionName.__name__)] = end - start

    return res, times



def generateMD5(self,text) -> str:
    return hashlib.md5(text).hexdigest()

def generateMD5Digest(self,text):
    return hashlib.md5(text).digest()


def generateSHA1(self, text) -> str:
    return hashlib.sha1(text).hexdigest()

def generateSHA1Digest(self, text):
    return hashlib.sha1(text).digest()

def generateSHA2(self, text) -> str:
    return hashlib.sha256(text).hexdigest()

def generateSHA2Digest(self, text):
    return hashlib.sha256(text).digest()

def generateSHA3(self, text) -> str:
    return hashlib.sha384(text).hexdigest()

def generateSHA3Digest(self, text):
    return hashlib.sha384(text).digest()
```

W celu implementacji poszczególnych funkcji skrótu wykorzystałem bibliotekę haslib. Funkcja generateAll odpowiada za zdobycie informacji na temat wykonywania się wszystkich funkcji zwracających hash w formacie heksadecymalnym.

```
def testCollision(self, encodedText,
numberOfTests,functionName,
numberOfTestingBytes):
    textLen = len(encodedText)
```

```
    numberOfCollisions = 0
    testingHash =
functionName(encodedText)[:numberOfTestingBytes]

    for i in range(1, numberOfTests + 1):
        randomString =
self.utils.generateRandomStringOfLength(textLen).
encode()
        randomHash =
functionName(randomString)[:numberOfTestingBytes]

        xorResult = bytes(a ^ b for a, b in
zip(testingHash, randomHash))

        if xorResult == b'\x00' *
numberOfTestingBytes:
            numberOfCollisions += 1

    return numberOfCollisions
```

Funkcja testująca kolizje sprawdza różnice na poszczególnych bitach hashów wygenerowanych dla numberOfTests losowych ciągów znaków oraz oryginalnego ciągu tym samym liczy liczbę kolizji na numberOfTestingBytes bitach.

```
def countBits(self,byteSeq):
    binaryRepresentation =
''.join(bin(byte)[2:].zfill(8) for byte in
byteSeq)
    numOnes = binaryRepresentation.count('1')
    numZeros = binaryRepresentation.count('0')
    return numZeros, numOnes

def bitChangeProbability(self,originalHash,
newHash):
    xorResult = bytes(a ^ b for a, b in
zip(originalHash, newHash))

    _, changedBits = self.countBits(xorResult)

    totalBits = len(originalHash) * 8
    changeProbability = changedBits / totalBits

    return changeProbability
```

```
def testSAC(self, encodedText, functionName,
iterations):
    originalText = encodedText
    originalHash = functionName(originalText)

    probabilities = dict()

    for i in range(iterations):

        encodedTextArray = bytearray(encodedText)

        randomPosition = random.randint(0,
len(encodedTextArray) - 1)
        randomBitIndex = random.randint(0, 7)

        encodedTextArray[randomPosition] ^= (1 <<
randomBitIndex)

        newText = bytes(encodedTextArray)
        newHash = functionName(newText)

        changeProbability =
self.bitChangeProbability(bytes(originalHash),
bytes(newHash))

        probabilities[i] = changeProbability

    return probabilities
```

Funkcja countBits zlicza wystąpienia zer oraz jedynek w podanej sekwencji bajtowej.

Funkcja bitChangePrrobability na podstawie operacji xor wylicza liczbę różnic w porownywanych sekwencjach oraz prawdopodobieństwo zmiany bitu.

Funkcja testSAC testuje efekt kaskadowy SAC dla podanej funkcji haszującej, modyfikując losowo pojedynczy bit w zakodowanym tekście i sprawdzając zmiany w wynikowym haszu. Przechowuje

prawdopodobieństwo zmiany bitów między oryginalnym a nowym haszem dla każdej iteracji. Na końcu zwraca słownik z wynikami analizy dla określonej liczby iteracji.

5. Rola soli w tworzeniu skrótów.

   Sól w tworzeniu skrótów pełni kluczową rolę w zabezpieczaniu haseł przed atakami. Dodanie unikalnej losowej wartości do każdego hasła zmniejsza skuteczność ataków słownikowych, dodana losowość sprawia, że nawet identyczne hasła mają różne skróty, utrudniając ich odgadnięcie. Uniemożliwia również identyfikację użytkowników na podstawie identycznych hashy, co zwiększa prywatność.
6. Czy funkcję MD5 można uznać za bezpieczną?

   Funkcja MD5 nie jest uznawana za bezpieczną, ponieważ jest podatna na kolizje, czyli sytuacje, w których różne dane wejściowe generują ten sam skrót. Już w 2004 roku zespół MD5CRK prowadzony przez Xiaoyun Wanga udowodnił skuteczność ataków kolizyjnych a dokładniej ataków urodzinowych na funkcję MD5. Dodatkowo, szybkość obliczeniowa funkcji MD5, zwłaszcza w dzisiejszych czasach znacznie ułatwia przeprowadzenie ataków typu „brute force”.