ECRYP PROJECT

SHA-3

Szymon Skalski
277081
ECRYP 20L

Introduction

This project is about SHA-3 function. This document contains overall inormation and theoretical introduction needed to understand the concept and basics of SHA-3. Apart from theory, there is practical implementation of the knowledge mentioned here – a program written in Python used to encode strings using SHA-3 functions.

Sha-3 – what is it?

Sha-3 is the member of the Secure Hash Algorithm family of cryptographic hash functions published by the National Institute of Standards and Technology. It is the latest member of the family, released in 2015. The purpose of SHA-3 is that it can be directly substituted for SHA-2 in current applications if necessary, and to significantly improve the robustness of NIST's overall hash algorithm toolkit.

Sha-3 bases on the approach called the sponge construction, in which data is "absorbed" into the sponge, then the result is "squeezed" out. In the absorbing phase, message blocks are XORed into a subset of the state, which is then transformed as a whole using a permutation function f. In the squeeze phase, output blocks are read from the same subset of the state, alternated with the state transformation function f. The size of the part of the state that is written and read is called the rate and the size of the part that is untouched by input/output is called capacity. The capacity determines the security of the scheme and the maximum security level is half the capacity value.

Sponge Construction

The sponge construction is a mode of operation, based on a fixed-length permutation (or transformation) and on a padding rule, which builds a function mapping variable-length input to variable-length output. Such a function is called a **sponge function**. It takes as input an element of Z_2^*, i.e., a binary string of any length, and returns a binary string with any requested length, i.e., an element of Z_2^* with **n** a user-supplied value. A sponge function is a generalization of both hash functions, which have a fixed output length, and stream ciphers, which have a fixed input length. It operates on a finite state by iteratively applying the inner permutation to it, interleaved with the entry of input or the retrieval of output.

In other words, sponge function is any of a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any desired length.

Sponge functions can be used to model or implement many cryptographic primitives, including cryptographic hashes, message authentication codes, mask generation functions, stream ciphers, pseudo-random number generators, and authenticated encryption.

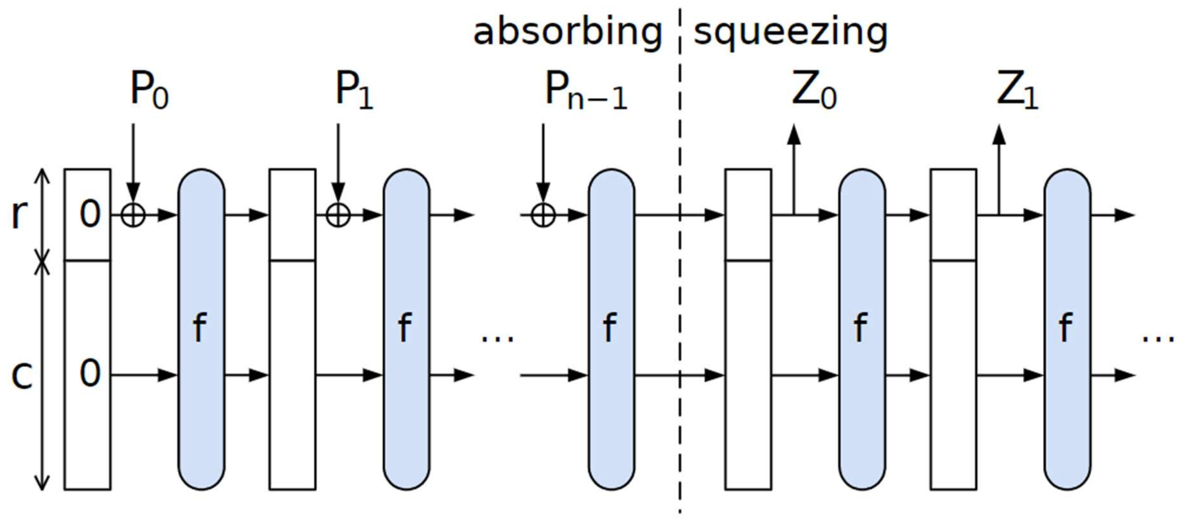


Fig. 1. The sponge construction for hash functions. Pi are blocks of the input string, Zi are hashed output blocks.

SHA-3's speed

The speed of SHA-3 hashing of long messages is dominated by the computation of f=Keccak-f[1600] and XORing S with the extended Pi, an operation on b=1600bits. However, since the last c bits of the extended Pi are 0 anyway, and XOR with 0 is a NOP, it is sufficient to perform XOR operations only for r bits. The lower r is, the less efficient but more secure the hashing becomes since fewer bits of the message can be XORed into the state before each application of the computationally expensive f.
All the above-mentioned data is clearly visible in the next table.

SHA-3 Instances

SHA-3 has four main instances: 224, 256, 384 and 512. Below, there is a table with comparison. There is also 16 other instances not mentioned here, but it is worth to notice that they exist.

Instance	Output size *d*	Rate *r* = block size	Capacity *c*	Security strengths in bits		
				Collision	Preimage	2nd preimage
SHA3-224	224	1152	448	112	224	224
SHA3-256	256	1088	512	128	256	256
SHA3-384	384	832	768	192	384	384
SHA3-512	512	576	1024	256	512	512

Security- quantum attacks

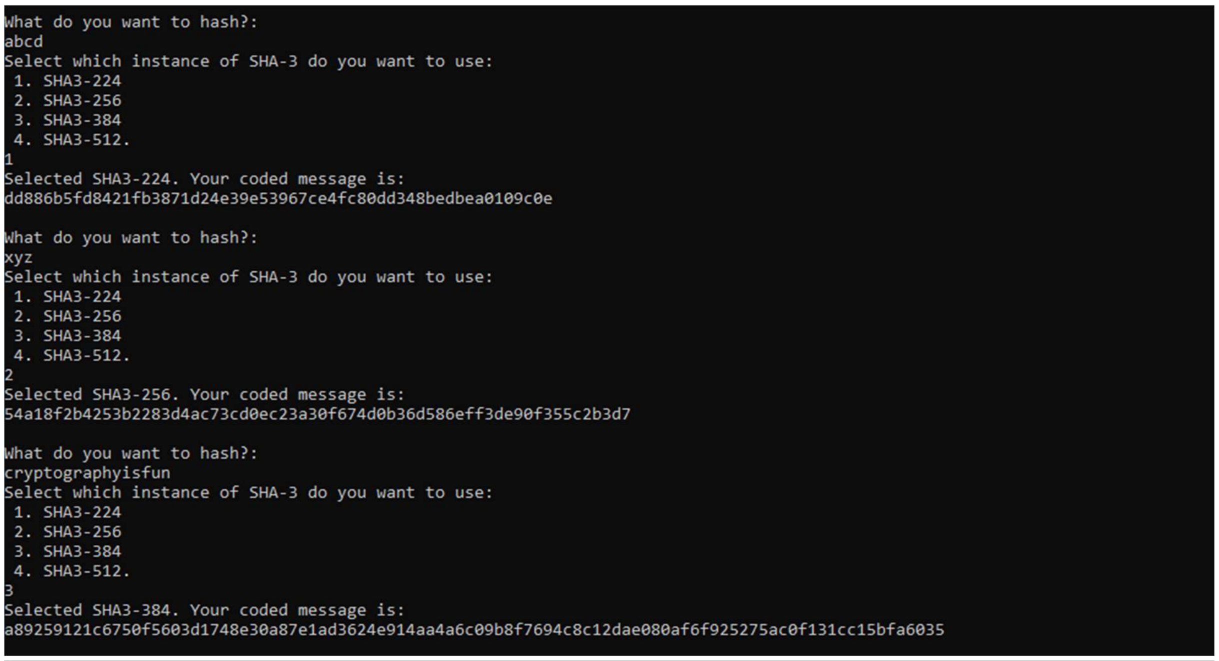
Grover's algorithm shows that there is a general result that quantum computers can perform a structured preimage attack in $\sqrt{2^d} = 2^{d/2}$, while a classical brute-force attack needs 2^d. A structured preimage attack implies a second preimage attack and thus a collision attack. A quantum computer can also perform a birthday attack*, thus break collision resistance, in $\sqrt[3]{2^d} = 2^{d/3}$. Noting that the maximum strength can be $c/2$ this gives the following upper bounds on the quantum security of SHA-3:

* An attack that can be used to abuse communication between two or more parties. It exploits the mathematics behind the birthday problem in probability theory.

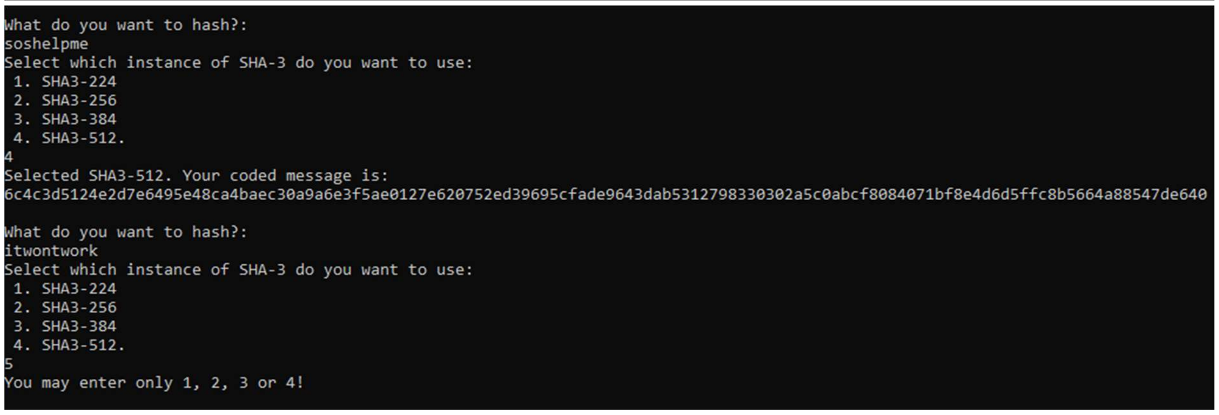
Instance	Security strengths in bits			
	Collision (Brassard et al.)	Collision (Bernstein)	Preimage	2nd preimage
SHA3-224	74⅔	112	112	112
SHA3-256	85⅓	128	128	128
SHA3-384	128	192	192	192
SHA3-512	170⅔	256	256	256

The program

To demonstrate the way in which SHA-3 works I wrote a program that lets the user to code their message using four different instances. It asks the user to write their desired message to be encrypted, select the desired function and then prompts the encrypted message on the screen. The screenshots from Python console of programs usage and the code are shown below.

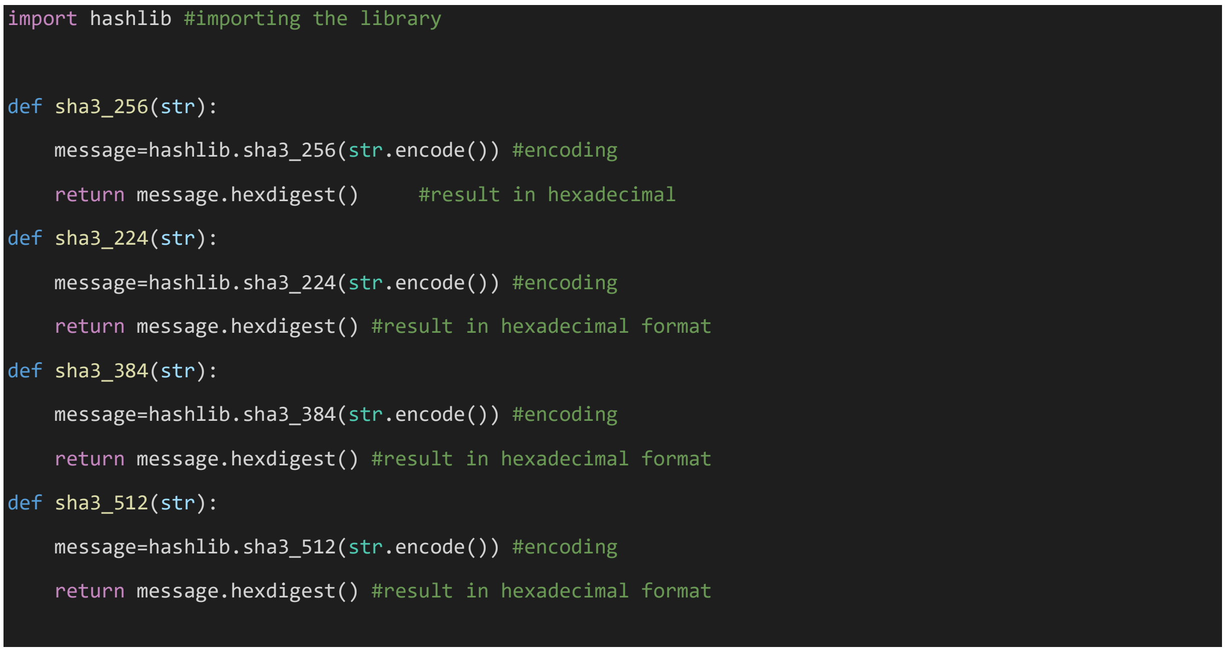


```
What do you want to hash?:
abcd
Select which instance of SHA-3 do you want to use:
 1. SHA3-224
 2. SHA3-256
 3. SHA3-384
 4. SHA3-512.
1
Selected SHA3-224. Your coded message is:
dd886b5fd8421fb3871d24e39e53967ce4fc80dd348bedbea0109c0e

What do you want to hash?:
xyz
Select which instance of SHA-3 do you want to use:
 1. SHA3-224
 2. SHA3-256
 3. SHA3-384
 4. SHA3-512.
2
Selected SHA3-256. Your coded message is:
54a18f2b4253b2283d4ac73cd0ec23a30f674d0b36d586eff3de90f355c2b3d7

What do you want to hash?:
cryptographyisfun
Select which instance of SHA-3 do you want to use:
 1. SHA3-224
 2. SHA3-256
 3. SHA3-384
 4. SHA3-512.
3
Selected SHA3-384. Your coded message is:
a89259121c6750f5603d1748e30a87e1ad3624e914aa4a6c09b8f7694c8c12dae080af6f925275ac0f131cc15bfa6035
```



```
What do you want to hash?:
soshelpme
Select which instance of SHA-3 do you want to use:
 1. SHA3-224
 2. SHA3-256
 3. SHA3-384
 4. SHA3-512.
4
Selected SHA3-512. Your coded message is:
6c4c3d5124e2d7e6495e48ca4baec30a9a6e3f5ae0127e620752ed39695cfade9643dab5312798330302a5c0abcf8084071bf8e4d6d5ffc8b5664a88547de640

What do you want to hash?:
itwontwork
Select which instance of SHA-3 do you want to use:
 1. SHA3-224
 2. SHA3-256
 3. SHA3-384
 4. SHA3-512.
5
You may enter only 1, 2, 3 or 4!
```

Python code



```python
import hashlib #importing the library


def sha3_256(str):
    message=hashlib.sha3_256(str.encode()) #encoding
    return message.hexdigest()      #result in hexadecimal
def sha3_224(str):
    message=hashlib.sha3_224(str.encode()) #encoding
    return message.hexdigest() #result in hexadecimal format
def sha3_384(str):
    message=hashlib.sha3_384(str.encode()) #encoding
    return message.hexdigest() #result in hexadecimal format
def sha3_512(str):
    message=hashlib.sha3_512(str.encode()) #encoding
    return message.hexdigest() #result in hexadecimal format
```

```python
def main():
    message = input ("What do you want to hash?: \n")
    decision=input("Select which instance of SHA-3 do you want to use:\n 1. SHA3-224 \n 2. SHA3-
256 \n 3. SHA3-384\n 4. SHA3-512. \n")
    if (decision=="1"):
        output = sha3_224(message)  #calling function, result calculation
        print('Selected SHA3-224. Your coded message is: \n' + str(output)) #printing coded output
    elif (decision == "2"):
        output = sha3_256(message)  #calling function, result calculation
        print("Selected SHA3-256. Your coded message is:  \n" +str(output)) #printing coded output
    elif (decision =="3"):
        output = sha3_384(message)
        print("Selected SHA3-384. Your coded message is: \n" +str(output))
    elif(decision == "4"):
        output = sha3_512(message)
        print("Selected SHA3-512. Your coded message is:  \n" +str(output))
    else:
        print("You may enter only 1, 2, 3 or 4! ")
        main()

main()
```