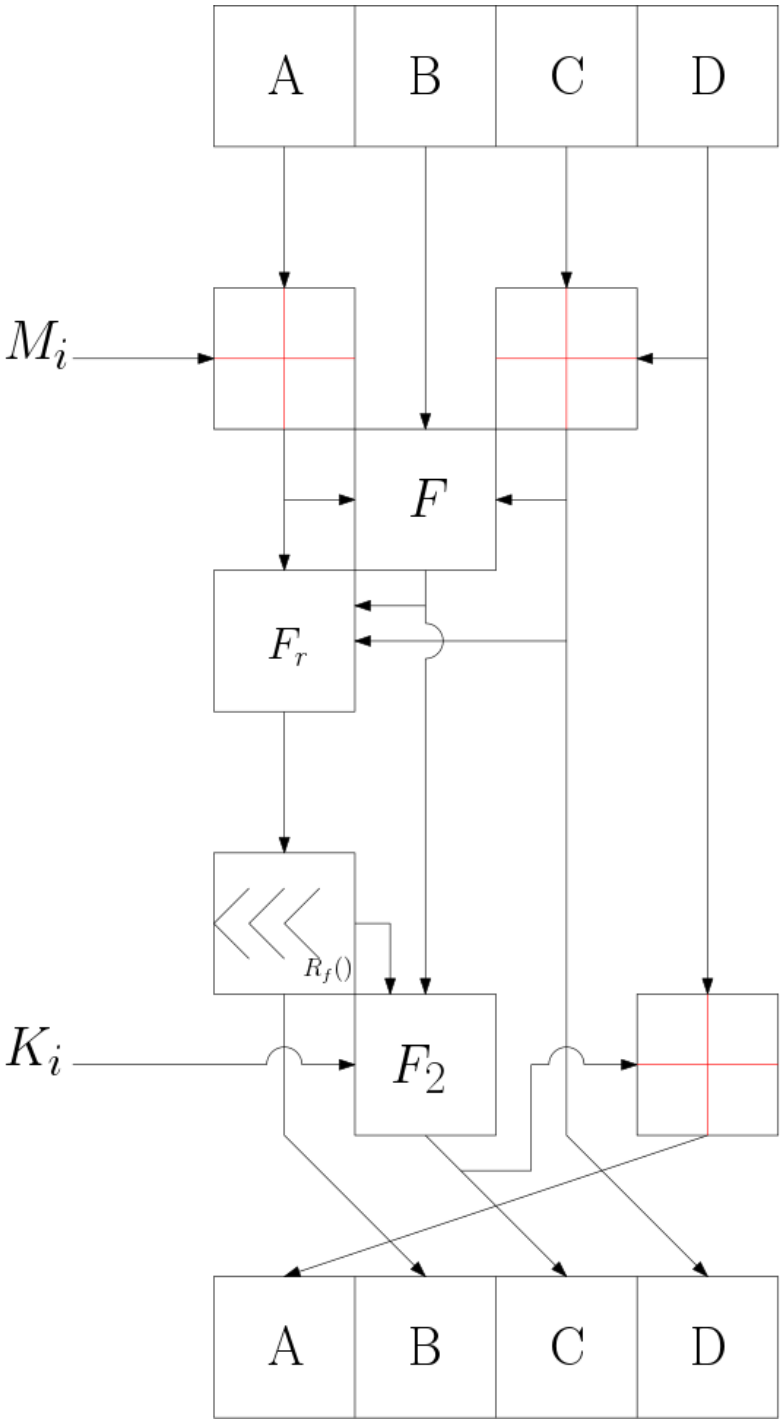
Cryptographic Hash Function:

Deliverable 1:



K_i

K_i is the operation constant, it's calculated at the beginning of every operation as the combination of a prime number and the state.

K_i = ((2926415965689092963) + ((A & ~B) ^ (C & D))) + ~A

F (Function 1)

F(A, B, C) = (A & B) ^ (A & C) ^ (B & C)

F2 (Function 2)

F2(A, B, K_i) = (A & B) ^ (~B & K_i)

Fr

This is the random function generator, for any operation of the hash it will perform the following operation

((A ^ ~B) & (C ^ B)) % 4

Depending on the result of this the function will be one of the following functions which were obtained from the SHA-2 wikipedia page.

- F(A, B, C) = (A & B) ^ (~A & C)
- F(A, B, C) = (A & B) ^ (A & C) ^ (B & C)
- F(A) = (A >> 2) ^ (A >> 13) ^ (A >> 22)
- F(C) = (C >> 6) ^ (C >> 11) ^ (C >> 25)

Rf

The number of bits that we rotate is randomly created as well using the following code.

A = (A >> (64 - ((A ^ K_i) % 64))) | (A << ((A ^ K_i) % 64)); // Barrel shift right (A ^ K_i) % 64 bits.

Initial Values

These are random ~64 bit prime numbers obtained from a prime number generator.

- A = 539958729876229229
- B = 881213617895827187
- C = 1619298787036835669
- D = 4571622384984713413

Deliverable 2:

Computationally Efficient:

Putting in a very large input (~78332 characters) there is no noticeable latency between hitting enter and getting the output.

Authenticity will be demonstrated during presentation.

Deterministic:

Our function does not depend on any random data at runtime. The only random data coming into the function is the input, and therefore it will always produce the same digest for the same message. Here is an example run.

Input: ./hash -p "Hello World"

Output: zEHW5LRVaC/Xs2MuRL3hnB8zZgJPvOH4IRai3lDI6VE=

Input: ./hash -p "Hello World"

Output: zEHW5LRVaC/Xs2MuRL3hnB8zZgJPvOH4IRai3lDI6VE=

Authenticity will be demonstrated during presentation.

Pre-image Resistant:

Finding a message that generates a given value would be very difficult, the search space for the output is 256 bit which is very large. Any change in inputs changes the output by ~50% when changing a single input bit. This means that inputs do not relate to outputs in an easily figured out way.

Running a couple of tests for demonstration…

INPUT: Hello World

INPUT: Iello World

OUTPUT: zEHW5LRVaC/Xs2MuRL3hnB8zZgJPvOH4IRai3lDI6VE=

OUTPUT: sEEX9ExXhi3YazOqPn//v9krE6o8r/+KhCjGnf8P9to=

bit_difference = 114

INPUT: a

INPUT: b

OUTPUT: rhpxaOwYjyRMPA+Ny9r10002AyRJaqXBWvTTSDQOjcM=

OUTPUT: jJxVz5aTV8v73bFv98f9I/nNOWXXxfkDSHw3rFmdMb8=

bit_diff = 131

Avalanche effect (related to Preimage Resistance)

Any small change in the input changes the input so much that they don't appear to be related.

INPUT: Test Message which is a bit longer than the previous ones

INPUT: Uest Message which is a bit longer than the previous ones

OUTPUT: uCf4yf8o92V64O6f9/XXnXrgz7f3Ncfnu2re1F936gE=

OUTPUT: 6T3sKKeh4mz3N13Nn/kvk/EzXeWT+yaD2TPtZfJupKk=

bit_diff = 114

INPUT: Test Message which is a bit longer than the previous oned

INPUT: Uest Message which is a bit longer than the previous oned

OUTPUT: 6uVqaLlAyfCnDfy/gw9G9aFp+L0BW0b1yY5IWfEDPqs=

OUTPUT: SaAWmh1tsM1fb+rPcnft1lfuqct6d/1WZUU5uVbjoSc=

bit_diff = 135

INPUT: Test Message which is a bit longer than the previous one

INPUT: Uest Message which is a bit longer than the previous one

OUTPUT: 2hnz8+QNEO2zN7vH/j4+x9EnmMUePj7HFMnG7Bb7upg=

OUTPUT: rxegtkFdcZxX82jmYOC+1U2rSuIuyL7R5o3N4oRT88c=

bit_diff = 111

INPUT: Test Message which is a bit longer than the previous one.

INPUT: Uest Message which is a bit longer than the previous one.

OUTPUT: q/WFtRQhg2VBNLc6cFoKmkE1cjnCegOKadhCVEt+XMU=

OUTPUT: que/UpHCEPmlx0/Sk/fDp6XCT9OUt8CgwF1SPvm89eg=

bit_diff = 131

It can be seen from the outputs that every time the input is changed there is an approximate 50% change in the output bits. By definition, this function achieves avalanche effect.