NAME: DEEP DHAKATE

CLASS: F.Y   DIV: A

Subject: MPBI, PPS Mini Project

Topic: Secure Hashing Algorithm Base 10 → Iconic Hashing

Documentation

1 | Hashing

1.1 | What is hashing?

Hans Peter Luhn invented Hashing. Hashing is the process of transforming any given key or a string of characters into another value. A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. A good hash algorithm should be complex enough such that it does not produce the same hash value from two different inputs. If it does, this is known as a hash collision. A hash algorithm can only be considered good and acceptable if it can offer a very low chance of collision. Hashing is relevant to -- but not limited to -- data indexing and retrieval, digital signatures, cybersecurity and cryptography. Hashing is most commonly used to implement hash tables.

1.2 | Why hashing algorithm necessary?

> Hashing gives a more secure and adjustable method of retrieving data compared to any other data structure. It is quicker than searching for lists and arrays. In the very range, Hashing can recover data in 1.5 probes, anything that is saved in a tree. Hashing, unlike other data structures, doesn't define the speed. A balance between time and space has to be maintained while hashing. There are two ways of maintaining this balance.

-->Controlling speed by selecting the space to be allocated for the hash table.

-->Controlling space by choosing a speed of recovery.

> Hashed passwords cannot be modified, stolen, or jeopardized. No well-recognized and efficient key or encryption scheme exists that can be misused. Also, there is no need to worry if a hash code is stolen since it cannot be applied anywhere else.

> Two files can be compared for equality easily through hashing. There is no need to open the two documents individually. Hashing compares them word-by-word and the computed hash value instantly tells if they are distinct. This advantage can be used for the verification of a file after it has been shifted to a new place. It is an example of Sync Back which is a file backup program.

> In DBMS, hashing is used to search the location of the data without using index structure. This method is faster to search using the short hashed key instead of the original value.

>Application of Hashing:

-->Password Verification

-->Compiler Operation

-->Rabin-Karp Algorithm

-->Data Structures

-->Message Digest

1.3 | Types of Hashes

There are many different types of hash algorithms such as RipeMD, Tiger, xxhash and more, but the most common type of hashing used for file integrity checks are MD5, SHA-2 and CRC32.

>MD5 - An MD5 hash function encodes a string of information and encodes it into a 128-bit fingerprint. MD5 is often used as a checksum to verify data integrity. However, due to its age, MD5 is also known to suffer from extensive hash collision vulnerabilities, but it's still one of the most widely used algorithms in the world.

>SHA-1: This is the second version of the Secure Hash Algorithm standard, SHA-0 being the first. SHA-1 creates 160-bit outputs. SHA-1 is one of the main algorithms that began to replace MD5, after vulnerabilities were found. SHA-1 gained widespread use and acceptance. SHA-1 was actually designated as a FIPS 140 compliant hashing algorithm.

>SHA-2 – SHA-2, developed by the National Security Agency (NSA), is a cryptographic hash function. SHA-2 includes significant changes from its predecessor, SHA-1. The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

>CRC32 – A cyclic redundancy check (CRC) is an error-detecting code often used for detection of accidental changes to data. Encoding the same data string using CRC32 will

always result in the same hash output, thus CRC32 is sometimes used as a hash algorithm for file integrity checks. These days, CRC32 is rarely used outside of Zip files and FTP servers.

>LANMAN: Microsoft LANMAN is the Microsoft LAN Manager hashing algorithm. LANMAN was used by legacy Windows systems to store passwords. LANMAN used DES algorithms to create the hash. The problem is that LANMAN's implementation of the DES algorithm isn't very secure, and therefore, LANMAN is susceptible to brute force attacks. LANMAN password hashes can actually be cracked in just a few hours. Microsoft no longer uses LANMAN as the default storage mechanism. It is available, but is no longer turned on by default.

>NTLM: This is the NT LAN Manager algorithm. The NTLM algorithm is used for password hashing during authentication. It is the successor of the LANMAN algorithm. NTLM was followed with NTLMv2. NTLMv2 uses an HMAC-MD5 algorithm for hashing.

Even though encryption is important for protecting data, sometimes it is important to be able to prove that no one has modified the data. This you can do with hashing algorithms. A hash is a one-way function that transforms data in such a way that, given a hash result (sometimes called a digest), it is computationally infeasible to produce the original message. Besides being one-way, hash functions have some other basic properties:

- They take an input of any length and produce an output of unpredictable sequence.
- They should be efficient and fast to compute.
- They should be computationally infeasible to invert.
- They should be strongly collision free.

2 | MD5 Hashing Algorithm

2.1 | What is Message Digest Algorithm?

The MD5 (message-digest algorithm) hashing algorithm is a one-way cryptographic function that accepts a message of any length as input and returns as output a fixed-length digest value to be used for authenticating the original message. Message digests, also known as hash functions, are one-way functions; they accept a message of any size as input and produce as output a fixed-length message digest.

MD5 is the third message-digest algorithm Rivest created. MD2, MD4 and MD5 have similar structures, but MD2 was optimized for 8-bit machines, in comparison with the two later algorithms, which are designed for 32-bit machines. The MD5 algorithm is an extension of MD4, which the critical review found to be fast but potentially insecure. In comparison, MD5 is not quite as fast as the MD4 algorithm, but offered much more assurance of data security.

How does MD5 work?

The MD5 message-digest hashing algorithm processes data in 512-bit strings, broken down into 16 words composed of 32 bits each. The output from MD5 is a 128-bit message-digest value.

Computation of the MD5 digest value is performed in separate stages that process each 512-bit block of data along with the value computed in the preceding stage. The first stage begins with the message-digest values initialized using consecutive hexadecimal numerical values. Each stage includes four message-digest passes, which manipulate values in the current data block and values processed from the previous block. The final value computed from the last block becomes the MD5 digest for that block.

2.2 | Why MD5?

Although originally designed as a cryptographic message authentication code algorithm for use on the internet. Ronald Rivest, founder of RSA Data Security LLC and professor at Massachusetts Institute of Technology, designed MD5 in 1991 as an improvement to a prior message-digest algorithm, MD4. Describing it in Internet Engineering Task Force (IETF) Request for Comments (RFC) 1321, "The MD5 Message-Digest Algorithm," he wrote:

The algorithm takes as input a message of arbitrary length and produces as output a 128-bit 'fingerprint' or 'message digest' of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be 'compressed' in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

2.3 | Vulnerabilities in MD5 Hash?

Although originally designed as a cryptographic message authentication code algorithm for use on the internet, MD5 hashing is no longer considered reliable for use as a cryptographic checksum because security experts have demonstrated techniques capable of easily producing MD5 collisions on commercial off-the-shelf computers. An encryption collision means two files have the same hash. Hash functions are used for message security, password security, computer forensics and cryptocurrency.

Collisions in the MD5 cryptographic hash function

It is now well-known that the crytographic hash function MD5 has been broken. In March 2005, Xiaoyun Wang and Hongbo Yu of Shandong University in China published an article in which they describe an algorithm that can find two different sequences of 128 bytes with the same MD5 hash. One famous such pair is the following:

```
d131dd02c5e6eec4693d9a0698aff95c2fcab58712467eab4004583eb8fb7f89

55ad340609f4b30283e488832571415a085125e8f7cdc99fd91dbdf280373c5b

d8823e3156348f5bae6dacd436c919c6dd53e2b487da03fd02396306d248cda0

e99f33420f577ee8ce54b67080a80d1ec69821bcb6a8839396f9652b6ff72a70
```

and

```
d131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb7f89

55ad340609f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5b

d8823e3156348f5bae6dacd436c919c6dd53e23487da03fd02396306d248cda0

e99f33420f577ee8ce54b67080280d1ec69821bcb6a8839396f965ab6ff72a70
```

Each of these blocks has MD5 hash 79054025255fb1a26e4bc422aef54eb4. Ben Laurie has a nice website that visualizes this MD5 collision. For a non-technical, though slightly outdated, introduction to hash functions, see Steve Friedle's Illustrated Guide.

Exploits

As we will explain below, the algorithm of Wang and Yu can be used to create files of arbitrary length that have identical MD5 hashes, and that differ only in 128 bytes somewhere in the middle of the file. Several people have used this technique to create pairs of interesting files with identical MD5 hashes:

Magnus Daum and Stefan Lucks have created two PostScript files with identical MD5 hash, of which one is a letter of recommendation, and the other is a security clearance.

Eduardo Diaz has described a scheme by which two programs could be packed into two archives with identical MD5 hash. A special "extractor" program turns one archive into a "good" program and the other into an "evil" one.

In 2007, Marc Stevens, Arjen K. Lenstra, and Benne de Weger used an improved version of Wang and Yu's attack known as the chosen prefix collision method to produce two executable files with the same MD5 hash, but different behaviours. Unlike the old method, where the two files could only differ in a few carefully chosen bits, the chosen prefix method allows two completely arbitrary files to have the same MD5 hash, by appending a few thousand bytes at the end of each file. (Added Jul 27, 2008).

Didier Stevens used the evilize program (below) to create two different programs with the same Authenticode digital signature. Authenticode is Microsoft's code signing mechanism, and although it uses SHA1 by default, it still supports MD5. (Added Jan 17, 2009).

An evil pair of executable programs

The following is an improvement of Diaz's example, which does not need a special extractor. Here are two pairs of executable programs (one pair runs on Windows, one pair on Linux).

Windows version:

hello.exe. MD5 Sum: cdc47d670159eef60916ca03a9d4a007

erase.exe. MD5 Sum: cdc47d670159eef60916ca03a9d4a007

Linux version (i386):

hello. MD5 Sum: da5c61e1edc0f18337e46418e48c1290

erase. MD5 Sum: da5c61e1edc0f18337e46418e48c1290

Reference: https://www.mscs.dal.ca/~selinger/md5collision/

3 | Iconic Hashing Algorithm 10

3.1 | What is IH?

IH is a unique hashing algorithm which works on base 10 digits for hashing. This hashing algorithm works on each of these digits to convert them into an unpredictable number format. The overall size of the hash is 38, 6 greater than the size of MD5 Hashing algorithm. The algorithm is designed such that operations made on the digits is almost impossible to reverse, and its really hard even for a machine to reverse it using all the possible digits. Each of letters, numbers, symbol, and even different number of spaces have different number of spaces.

“A” = 68024602468024602468024602446802778924

“B” = 91357035791357035791357035468024700357

“1” = 89249113835891613351670138356838923506

“2” = 11260903013891421252460136434856823586

“ ” = 23579166722357916672235791667223568236

“   ” = 90224457914680257914680257914680224466

3.2 | Unique digits in IH

In IH each of the digits are represented by unique digits. And the combination of all this digits for a string is called cipher text.

upper_case = [["A", "01"], ["B", "02"], ["C", "03"], ["D", "04"], ["E", "05"], ["F", "06"], ["G", "07"], ["H", "08"], ["I", "09"], ["J", "11"], ["K", "22"], ["L", "33"], ["M", "44"], ["N", "55"], ["O", "66"], ["P", "77"], ["Q", "88"], ["R", "99"], ["S", "12"], ["T", "13"], ["U", "14"], ["V", "15"], ["W", "16"], ["X", "17"], ["Y", "18"], ["Z", "19"]]

lower_case = [["a", "23"], ["b", "24"], ["c", "25"], ["d", "26"], ["e", "27"], ["f", "28"], ["g", "29"], ["h", "34"], ["i", "35"], ["j", "36"], ["k", "37"], ["l", "38"], ["m", "39"], ["n", "45"], ["o", "46"], ["p", "47"], ["q", "48"], ["r", "49"], ["s", "56"], ["t", "57"], ["u", "58"], ["v", "59"], ["w", "67"], ["x", "68"], ["y", "69"], ["z", "78"]]

number = [["1", "601"], ["2", "602"], ["3", "603"], ["4", "604"], ["5", "605"], ["6", "606"], ["7", "607"], ["8", "608"], ["9", "609"], ["0", "610"], [" ", "00"]] symbols = [["~", "1000"], ["`", "1001"], ["!", "1002"], ["#", "1003"], ["$", "1004"], ["%", "1005"], ["^", "1006"], ["&", "1007"], ["*", "1008"], ["(", "1009"], [")", "1010"], ["-", "1011"], ["_", "1012"], ["=", "1013"], ["+", "1014"], ["{", "1015"], ["}", "1016"], ["[", "1017"], ["]", "1018"], [":", "1019"], [";", "1020"], ['"', "1021"], ["'", "1022"], ["|", "1023"], ["\\", "1024"], ["<", "1025"], ["@", "1026"]]

The above mentioned 2D array represents the numbers with which upper case letters, lower case letters, numbers and symbols are represented.

```
def encode(string):
    upper_case = [["A", "01"], ["B", "02"], ["C", "03"], ["D",
"04"], ["E", "05"], ["F", "06"], ["G", "07"], ["H", "08"], ["I",
"09"], ["J", "11"], ["K", "22"], ["L", "33"], ["M", "44"], ["N",
"55"], ["O", "66"], ["P", "77"], ["Q", "88"], ["R", "99"], ["S",
"12"], ["T", "13"], ["U", "14"], ["V", "15"], ["W", "16"], ["X",
"17"], ["Y", "18"], ["Z", "19"]]
    lower_case = [["a", "23"], ["b", "24"], ["c", "25"], ["d",
"26"], ["e", "27"], ["f", "28"], ["g", "29"], ["h", "34"], ["i",
"35"], ["j", "36"], ["k", "37"], ["l", "38"], ["m", "39"], ["n",
"45"], ["o", "46"], ["p", "47"], ["q", "48"], ["r", "49"], ["s",
"56"], ["t", "57"], ["u", "58"], ["v", "59"], ["w", "67"], ["x",
"68"], ["y", "69"], ["z", "78"]]
    number = [["1", "601"], ["2", "602"], ["3", "603"], ["4",
"604"], ["5", "605"], ["6", "606"], ["7", "607"], ["8", "608"],
["9", "609"], ["0", "610"], [" ", "00"]]
    symbols = [["~", "1000"], ["`", "1001"], ["!", "1002"], ["#",
"1003"], ["$", "1004"], ["%", "1005"], ["^", "1006"], ["&", "1007"],
["*", "1008"], ["(", "1009"], [")", "1010"], ["-", "1011"], ["_",
"1012"], ["=", "1013"], ["+", "1014"], ["{", "1015"], ["}", "1016"],
["[", "1017"], ["]", "1018"], [":", "1019"], [";", "1020"], ['"',
"1021"], ["'", "1022"], ["|", "1023"], ["\\", "1024"], ["<",
"1025"], ["@", "1026"]]

    global cipher
    cipher = ""
    for char in string:
        for i in range(len(upper_case)):
            if char == upper_case[i][0]:
                cipher += str(upper_case[i][1])
            if char == lower_case[i][0]:
                cipher += str(lower_case[i][1])
        for t in range(len(number)):
                if char == number[t][0]:
                    cipher += number[t][1]
        for o in range(len(symbols)):
                if char == symbols[o][0]:
                    cipher += symbols[o][1]
    print(cipher)
    padding(cipher)
```

3.3 | Padding

Later the cipher text generated by the unique numbers are padded according to their length. If the length of cipher text is less than 64 then the code will embed 1-9 numbers at the end of the cipher text. If the length of cipher text is greater than 64, then the code will embed 1-9 at the end of cipher text till it attends the next nearest number which is divisible by 32. And then the cipher array us divided into 32 numbers, more over the last and the first entries of the cipher array of each 32 numbers are exchanged. Then the code generates 3 different keys to hash the numbers.

```
def padding(cipher):
    length = len(cipher)
    if length < 64:
        t = 1
        for i in range(length, 64):
            if t<10:
                cipher += str(t)
                t = t + 1
            if t>=10:
                t=1
    if length > 64:
        l =1
        for i in range(32):
            if len(cipher)%32!=0:
                if l >= 10:
                    l = 1
                cipher += str(l)
                l = l+1
            if len(cipher)%32==0:
                break

    cipher_array = []
    for char in cipher:
        cipher_array.append(char)
    split = []
    div = len(cipher)//32
    n = 0
    for m in range(div):
        t = ""
        for i in range(n, 32 + n):
            t = t + str(cipher_array[i])
        n += 32
        split.append(t)
    leng = len(split)
    temp = split[0]
    split[0] = split[leng - 1]
    split[leng - 1] = temp
    n_cipher = ""
    for i in range(len(split)):
        n_cipher += split[i]
    cipher_array = []
    for char in n_cipher:
        cipher_array.append(char)
    key_gen(cipher_array)
    key_gen2(cipher_array)
    key_gen3(cipher_array)
    hash(key, key2, key3, cipher_array)
```

3.3 | Key generation

First key is generated by taking twice the sum of all the numbers generated by hash.

For example, if cipher array is 1234567890 it will take sum of all the numbers two times. i.e 1st sum will be 45 and the second sum will be 4+5 = 9, Therefore the key generated is 9

The second key generated by taking the middle number of the cipher array.

Example: if cipher array is 123456789, 5 is the second key.

The third key is generated by taking the sum of first and last number.

Example: if cipher array is 123456789, 10 is the third key.

```
def key_gen(new_cipher):
    sum = 0
    for i in range(len(new_cipher)):
        sum += int(new_cipher[i])
    global key
    key = 0
    for char in str(sum):
        key += int(char)
    return
def key_gen2(new_cipher):
    length = int(len(new_cipher))
    global key2
    if length%2==0:
        key2 = new_cipher[int(length/2)]
    else:
        key2 = new_cipher[(length+1)/2]
    return key2
def key_gen3(new_cipher):
    length = len(new_cipher)
    global key3
    key3 = new_cipher[0]+new_cipher[length - 1]
    return
```

3.4 | Hash generation

Generation of hash: The first step involves the sum of each single numbers present in cipher array and divide it by 11. The reminder is stored in an array. The further hash generation is done on reminders. Then each reminder are added to the product of second key and the third key. Again, each of them is divided by 11 and the remainder is stored inside a hashing array. Then the numbers from 1-9 are appended to this hash array till the length of this array is not divisible by 38. Then the array is divided into 38 and each of 38 numbers are added to another 38 numbers array to each other by taking its modulus with 10. Resulting generated numbers is IH hash.

```
def hash(key, key2, key3, cipher_array):
    form = ""
    print(key)
    print(key2)
    print(key3)
    hash = ""
    for i in range(len(cipher_array)):
        temp = (int(cipher_array[i])+int(key))%11
        form += str(temp)
        temp = 0
    for i in range(len(form)):
        var = (int(form[i])+(int(key3)*int(key2)))%11
```

```python
        hash += str(var)
    format(hash)
def format(hash):
    l = 1
    for i in range(38):
        if len(hash)%38!=0:
            if l >= 10:
                l = 1
            hash += str(l)
            l = l+1
        if len(hash)%38==0:
                    break
    split = []
    a = ""
    for i in range(len(hash)):
        if (i+1)%38==0:
            a = a + hash[i]
            split.append(a)
            a = ""
        else:
            a = a + hash[i]

    t = ""
    for i in range(38):
        sum = 0
        for m in range(len(split)):
            k = split[m][i]
            sum = int(sum) + int(k)
        no = sum%10
        t = t + str(no)
    print(t)
    print(len(t))
```

3.5 | Code Output

```
Enter a string to hash: MyP@ssw0rd   → String to hash

44697710265656676104926  → Cipher text in base 10 format

79143522569468680546702597965159715814  → Hash of the string
```

```
Enter a string to hash: Prajwal@123

774923366723381026601602603

108000746690671215788414602141605209 81
```

3.6 | Usage

The algorithm is easy to use and is very user friendly. When a user runs a code in python language it asks user to input a string. This string is latter hashed by the algorithm. User just has to input the string which he has to hash.

Linux Operating System

```
git clone https://github.com/InTruder-Sec/IHA

cd IHA

python3 hash.py
```

Windows Operating System And MAC Operating System

https://github.com/InTruder-Sec/IHA

```
visit the above site, download the repository. Unzip the file and run the
python file names as hash.py.
```

An organisation can use **bottle.py, Flask, CherryPy, Pyramid, Django and web2py** to host a web application in python and include this python file for hashing of a password. Latter inside a login page they can us this python file to compare the hash of password stored inside a database.

4 | Security of Iconic Hashing algorithm

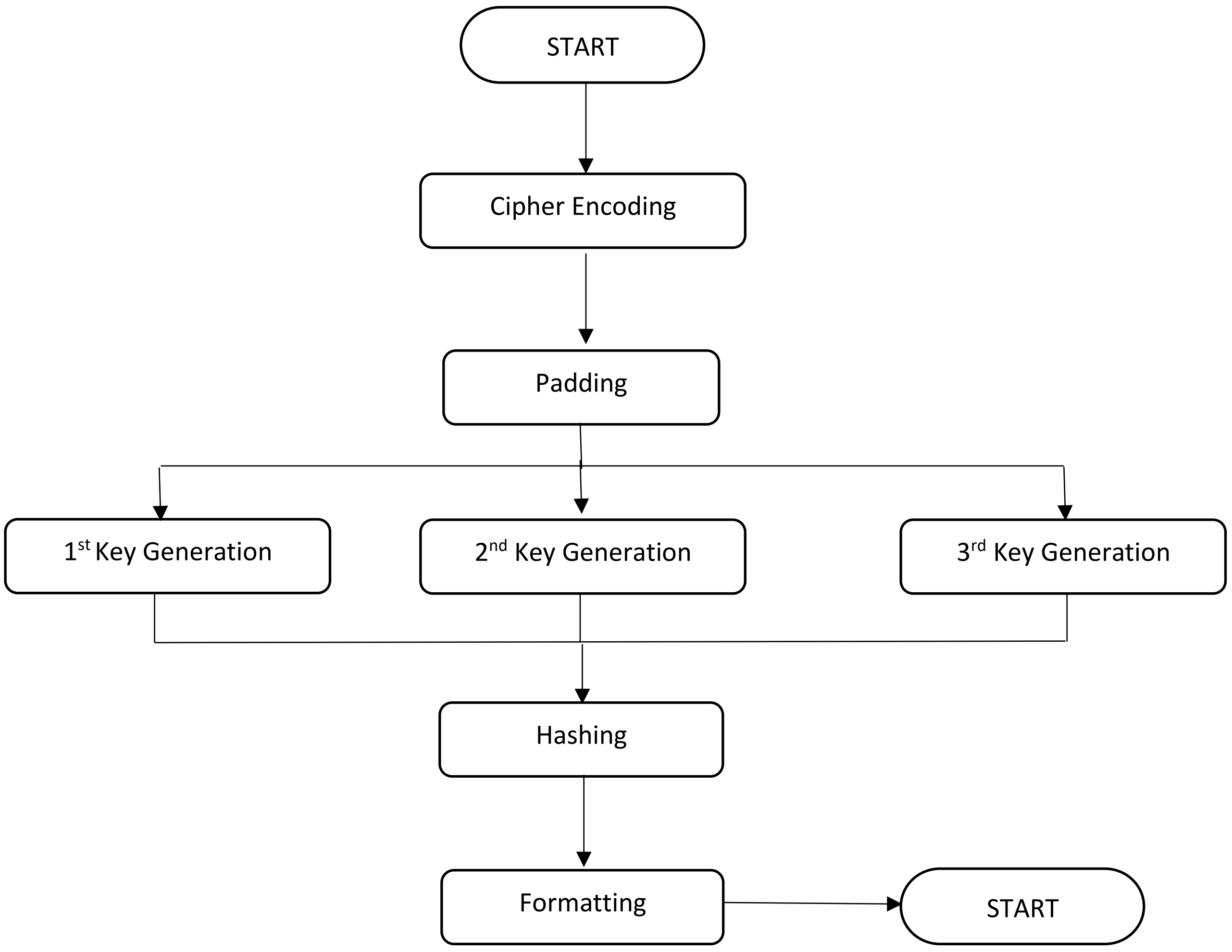
4.1 | Why IHA?

Iconic Hashing algorithm works on some unique digits. More over the three keys changes with respect to the string inputs so even if the user tries to craft a payload for 2 strings having same Hash, their keys would differ so that the operations made on its cipher text would differ. In order to have to hashes same It should have same cipher text as well as same keys to encrypt it. Hence it is almost impossible for a machine to reverse the hash or to detect two strings having same hash.

4.2 | MD5 vs IHA

MD5 is most used hashing algorithm even though it has some vulnerabilities. Many strings are already present which has same MD5 hash. Whereas in IHA it is rarely possible to have same hash of two strings. This is because MD5 works on the binary digits i.e 1,0. The string is converted into binary digits in MD5 hash and algorithm works on these binary digits making bits and resulting a 32-length hash string. In IHA the string is converted into unique digits in base 10 format i.e it involves numbers from 0 to 9. The IHA works on this unique base 10 cipher text to o convert it into hash of length 38, thus making it difficult to have two strings having same hash. Thus, collision of hashes is prevented in this algorithm. It is impossible to reverse the MD5 Hash, similarly it is almost impossible to reverse the IHA hash as it works on the principle of unique keys and modulus function.

5 | Flow Chart



6 | References

Iconic Hashing algorithm repository: https://github.com/InTruder-Sec/IHA

MD5 hashes: https://en.wikipedia.org/wiki/MD5