**Symmetric Hash-Chain-Encryption Protocol**

**Abstract**

    The purpose of this study is to demonstrate the effective use cases and theoretical security of Hash-Chain-Encryption protocol utilizing commonly known undecipherable encryption techniques such as Vernam Cipher. Apart from encryptions such as AES, which is quantum secure and has many practical uses, Vernam cipher lacks practicality but enforces an unbreakable communication barrier once successful. At some point, although not soon, computers and quantum computers can potentially surpass methods of brute force evaluating for the correct keys, examples of where technology has weakened or trivialized strong cryptographic algorithms such as RSA and AES by methods such as Grover’s algorithm and Shor’s Algorithm. This proposed protocol does not replace standard protocols and encryptions as it has its limitations albeit far less than the original Vernam cipher. The essential functionality of Hash-Chain-Encryption is building upon the Vernam cipher with multiple reuses without ever breaking the OTP (One Time Pad). Then, discovering the possible use cases and enforcements in practical applications describing its processes, finally the possible vulnerabilities and issues utilizing the protocol if not taken into consideration.

**Methodology**

    Hash-Chaining is the method of one-way functions hashing its values recursively producing a new and unique hash in its place. Hashing a function 3 times is represented as h3(func). There is no definitive way to undo a hash and decipher the original value, methods such as hash tables and rainbow tables are used when discovering a possible hash value in password stored databases. Salting is an extra technique used for passwords to decouple hashes and prevent an attacker from discovering multiple accounts with a common stored hash. For security purposes and maintaining the integrity of hashes, Hash-Chain-Encryption will use SHA-512 to prevent hash collisions and birthday attacks, with the probability table SHA-512[REF] has an extremely low rate of collision of probability p=0.75 given an occurrence of 1.9×1077 hashes. This is a prerequisite for understanding that Hash-Chaining is a secure operation that cannot be broken unless the amount of hash attempts exceeds the probability count or if the hashed document is already stored in a table, the latter may be prolonged indefinitely when proper salting and function hash parameter is longer than an arbitrary amount, such as greater than 8 characters for password/key hash requirements. The longer the function parameter, the harder it is to store the hash in a table for a dictionary attack.

    Hash-Chaining for the (HCE)Hash-Chain-Encryption will be like the defined Hash-Chaining mentioned except that the hashes will be appended to each other at the end. As an example:

Original: Text

Hash1: h(Text)

Hash2: h(h(Text)) || h(Hash1)

Hash3: h(h(h(Text))) || h(Hash2)

Hash4: h(h(h(h(Text)))) || h(Hash3)

Result: Hash1 + Hash2 + Hash3 + Hash4

      With regards to all other security mentioned to make sure that the hashes are not compromised. The primary/origin key should always be kept hidden, and a function generator such as SHA-512 will produce the string h(origin) + h2(origin) + … + hn(origin) from the origin, and the sum of strings from the origin will be called the secondary key. The origin will be made up of 512 bits/128 characters (CSRNG or TRNG) and utilize hashing algorithm SHA-512 to prevent hashing collisions generating hashes of 512bits/128 characters. The produced string of data will serve as the secondary key and origin as a primary key. The number of hashes needed for the secondary key are arbitrary and do not add extra security as the algorithm explicitly states that the first 512bits/128 characters hashed will make up the subsequent 512bits/128-character hash. The purpose of hashing 4 times on the origin key is to keep consistency when chaining the secondary key (4 hashes = 512chars. /2048 bits) and ease of use when hash-chaining. Smaller key length sizes such as 128(1 hash) or 256(2 hashes) are acceptable for the secondary key. Storage of the primary key is necessary and will act as our private key, as the secondary key can be produced from the primary key. Two separate modes exist where the secondary key does not need to be chained for smaller and simpler messages and block-chaining for larger files and format types.

[Simple Control Mode]

**Primary:** **origin, length = 128 chars.**

**Secondary: [hash1, hash2, hash3, hash4], length = 512 chars.**

**Message: If M < 512 chars. Append Salt until Message length = 512 chars.**

**ENCR: Message XOR Secondary, length = 512 chars.**

[Block-Hash-Chaining Mode]

**Primary:** **origin, length = 128**

**Secondary: [hash1, hash2, hash3, hash4],**

**Message: If M >= 512 chars. Append Salt until Message length % 512 = 0**

The secondary key will always be divisible by 512-character length for this instance. The appended salt on the plain text will contain /[EXT:1234567890….57453]/ of pseudo random numbers [CSRNG/TRNG] until the size length reaches % 512 = 0. The original message will be padded instead of the secondary key to make sure the message length and key length are of equal length. The secondary key and message XOR with each other producing the ciphertext that is theoretically undecryptable without knowing the message or the key. When the receiver receives the incoming ciphertext alongside an HMAC of the modified plaintext, it allows for auditing the signature after decrypting with another XOR operation utilizing the receiver's private key since both parties start off with the same private key. Using the secondary key will notify if the ciphertext or HMAC has been altered mid transit, requesting the sender to send the Ciphertext and HMAC again until successful signatures have been verified. Repeated attacks altering the packet information of the Cipher Text or the HMAC can indefinitely delay communication until a different mode of transportation is secured. Although attacks can be made against the methods of transportation, the message remains undecryptable regardless.

    The architecting feature representing Vernam Cipher is the XOR operation. When performed correctly the XORed bits become impossible to recover without either operand. Considerations to this are the famous OTP rules stating:

1. The key cannot be used more than once.

2. The key length must be equal to the message length.

3. The key must be truly random.

4. The key must be kept secret.

    1. HCE strictly updates the key using either [CSRNG/TRNG] for generating a new key, it is the user’s choice whether to hold and collect keys (To view complete conversations) on their trusted computer systems or the suggested approach of allowing the new private key to replace the old private key. By this means it prevents entire logs of communications from being breached on the system and continues a one-way asynchronous flow of data exchange to occur. Given the randomness of [CSRNG/TRNG] it is no easy number to guess and scoff at. The origin key is used once when sending the newly encrypted key, if during an interrupt or intercept the origin key may send the same newly generated key or generate another new one, there is no difference in security with either decision.

    2. HCE secondary keys always remain consistent with their length sizes; modification of the original message helps preserve attacks against the hash of the plaintext especially if messages are very short such as ‘Hi’ or ‘Hello’ to prevent tables for common phrases. On the last occurrence of the salt which is defined as: /[EXT:1245..97367]/ used as padding to the plaintext to meet equal length criteria. Upon decryption the padding is removed from the plaintext leaving the original unformatted message.

    3. HCE allows any means of number generation although for true impenetrable encryption requires TRNG which can be generated through many other applications such as environmental factors. For practical uses CSRNG is sufficient against computational and humane guessing, if the distribution of bits/characters/numbers remain relatively even, to prevent analysis of the number generator.

    4. HCE keys are stored in the computer system, whether at Application level or at the Operating System. Personal encrypted USBs can store the key for more security if there is distrust in the system. Storage of key is based on personal preference, by default the protocol will store the key in the OS such as every other private key that is stored in a personal computer. The key used in the application or OS can be encrypted again before storing, utilizing a master encryption application or a 2FA (Two-Factor Authentication) system to prevent internal systems or vulnerabilities on the application or OS from breaching the private messages.

**Problems:**

    Problem level: Convenience - **High**: The original issue as to why Vernam Cipher is not popularly used, is that the key will most certainly require 2 individuals to physically meet up in person to begin the secret exchange. The advantage of this protocol against Vernam is that there is no longer a need to continuously meet up in person to exchange new keys and new secret messages. HCE is a once and done deal, other semi secure methods such as RSA can exchange the private key initially, but any attacks on RSA will leave the private key vulnerable. HCE's recommended method is to physically exchange the keys in person through a secure USB, once that exchange starts through electronic communication there will be no need for the USB and onto the system or to keep the updated keys on the USB for further conversations.

    Problem level: Encryption - **Low**: Extremely large files can lead to a possibility of hash collision for SHA-512, to get a probable hash collision would be considered very improbable. If a hash collision did occur unintentionally this could lead a portion of texts/images to ciphertext attacks as a pattern could be made out and leaking parts of the text. If the possibility of a successful ciphertext attack occurred because of a hash collision it is a very possible outcome that the pattern or previous hash would lead to the private key breaking the protocol. Luckily HCE is made for communication exchange instead of data/storage exchange so it would be very improbable to impossible for a collision to occur. Storages can be encrypted and exchanged with HCE as terabytes of data are still too unlikely to cause a hash collision in practice but not thoroughly tested yet.

    Problem level: Encryption - **Low**: If a certain breach were to occur on the affected device and the keys are exposed unencrypted, the attacker will be able to breach the messages obtaining the private keys. This problem is not isolated to HCE but all systems that do not encrypt their keys before storing them. These targeted sophisticated attacks are unlikely but possible, to mitigate or nullify these attacks utilizing Vernam Cipher as an unbreakable barrier between the user and the device from the attacker or using a more common and convenient way such as an encryption manager or 2FA as mentioned in OTP rules #4.

    Problem level: Convenience - **High**: Addition or removal of another party would also be very tricky and complicated. Sending other parties the private key is doable but HCE's recommendation is to start a new conversation with new keys and exchange the key with whom the conversation will belong to in person.

Problem level: Implementation - **High**: The keys for both parties must always be the same, there cannot be 2 senders or 2 receivers at a given time, this would cause a misalignment to both party’s keys. Sending would need to be verified by which instance was sent first then the receiver would need to receive the data updating the key before being able to send. On a similar note, there should **NEVER** be 2 messages sent encrypted with the same secondary key, this would cause the algorithm to fall apart as it breaks the rules of OTP.

**Algorithm: Simple Control Mode**

Hashes: Sha-512

Encrypt: XOR

Decrypt: XOR

Assume both parties securely have the same Private Key (Physical Exchange) Key1

**Sender Side:**

input = RAW TEXT

Message M1 = (input) size: (512 char. Length) (Last characters reserved for Salt).

Key1 priv. = Shared key, must be securely transferred. (512 char. Length).

Hash1 = Hash (Key1). (128 char. Length).

Hash2 = Hash (Hash1). (128 char. Length).

Hash3 = Hash (Hash2). (128 char. Length).

Hash4 = Hash (Hash3). (128 char. Length).

Key2 priv. = Hash1 + Hash2 + Hash3 + Hash4. (512 char. Length).

Authentication A1 = Hash(M1). (128 char. Length).

Ciphertext CT = Key2 ⊕ M1. (512 char. Length).

Keynew = CSPRNG|TRNG. (128 char. Length).

Authentication A2 = Hash (Key1 + Keynew). (128 char. Length).

KeySecret = Key1 ⊕ Keynew. (128 char. Length).

Discard Key1 and replace it with Keynew. Only after Receiver has successfully received matching signatures or Save Key1 locally on a USB. Preferred use case.

Key1 <- Keynew. Discard old with new.

SEND (CT, A1, KeySecret, A2).

**Receiver Side:**

RECEIVE (CT, A1, KeySecret, A2).

Decrypted Message MDecr = CT ⊕ Key2. (512 char. Length).

Authentication A3 = Hash (MDecr). (128 char. Length).

IF A3 == A1, Valid, Else tampered request data again.

Original Text OT = Parse Salt from M1. (Input Length).

KeyUpdate = Key1 ⊕ KeySecret. (128 char. Length).

Authentication A4 = Hash (KeyUpadate). (128 char. Length).

IF A4 == A2, Valid, Else reject key and DO NOT UPDATE, request data again.

Key1 <- KeyUpdate, Discard Key or store locally.

END.

**Algorithm: Block-Hash-Chaining-Mode**

Hashes: Sha-512

Encrypt: XOR

Decrypt: XOR

Assume both parties securely have the same Private Key (Physical Exchange) Key1

**Sender Side:**

input = RAW TEXT

Message M1 = (input) size: (N char. Length)

M1 = M1 + Salt (Last characters reserved for Salt % 512 = 0).

ML = (Message Length N >= 512 and 512 % = 0)

Key1 priv. = Shared key, must be securely transferred. (512 char. Length).

Hash1 = Hash (Key1). (128 char. Length).

Hash2 = Hash (Hash1). (128 char. Length).

Hash3 = Hash (Hash2). (128 char. Length).

Hash4 = Hash (Hash3). (128 char. Length).

Key2 priv. = Hash1 + Hash2 + Hash3 + Hash4. (512 char. Length).

Key2 priv. ext:1 = Hash4 + Hash5 + Hash6 + Hash7. (512 char. Length).

…

Key2 priv. ext: ML/512 = HashML/512-3 + HashML/512-2 + HashML/512-1 + HashML/512. (%512=0 char. Length).

Key2 priv. = Key2 priv + Key2 priv. ext:1 + … + Key2 priv. ext: ML/512

Authentication A1 = Hash(M1). (128 char. Length).

Ciphertext CT = Key2 ⊕ M1. (%512=0 char. Length).

Keynew = CSPRNG|TRNG. (128 char. Length).

Authentication A2 = Hash (Key1 + Keynew). (128 char. Length).

KeySecret = Key1 ⊕ Keynew. (128 char. Length).

Discard Key1 and replace it with Keynew. Only after Receiver has successfully received matching signatures or SaveKey1 locally on a USB.

Key1 <- Keynew. Discard old with new.

SEND (CT, A1, KeySecret, A2).

**Receiver Side:**

RECEIVE (CT, A1, KeySecret, A2).

Decrypted Message MDecr = CT ⊕ Key2. (N char. Length, Length is % 512 = 0).

Authentication A3 = Hash (MDecr). (128 char. Length).

IF A3 == A1, Valid, Else tampered request data again.

Original Text OT = Parse Salt from M1. (Input Length).

KeyUpdate = Key1 ⊕ KeySecret. (128 char. Length).

Authentication A4 = Hash (Key1 + KeyUpadate). (128 char. Length).

IF A4 == A2, Valid, Else reject key and DO NOT UPDATE, request data again.

Key1 <- KeyUpdate Discard Key or store locally.

END.

**Conclusion**

    HCE is a considerably fast encryption protocol as it only utilizes 2 core principles, Hashing (SHA-512) which is very fast on x86 systems and the XOR operation which is one of the easiest fundamental operations in computer hardware that can be performed, faster than simple arithmetic. This protocol can be used in wide scaled messaging systems, but it will forfeit its security of transfer of the initial private key instantiated by the sole discretion of the provider hosting the system, also whether the conversation log would be kept retaining previous old keys in a cloud database which would be largely cumbersome to the cloud storage and be a primary target for attacks since attacking the encrypted files would be completely pointless. There is more optimism when looking at (E2E) End-to-End encryption performing encryption on the user side and saving data locally on the user device rather than in a database. It’s ideal use case would be the highest TOP SECRET and CONFIDENTIAL information that needs to be relayed multiple times to a single party securing data through transit whether it be a message or any type of file/image. For security minded individuals this could be overkill or exactly what is needed as transferring data will not be capable of being breached, ever. As with Vernam Cipher during very secretive missions during war time this form of communication was necessary but difficult as meeting physically after every secret exchange was not always possible to make. HCE capitalizes on limited but high-stake information, and generally does not have its place in common messaging systems except for those who are privacy and security minded.