Toha Key Hardened Function

Ahmad Almorabea[1]

1 *ahmad@almorabea.net*

Abstract:

TOHA is Key Hardened Function designed in the general spirit of sequential memory- hard function which based on secure cryptographic hash function, the idea behind its design is to make it harder for an attacker to perform some generic attacks and to make it costly as well, TOHA can be used for deriving keys from a master password or generating keys with length of 256-bit to be used in other algorithm schemes, general approach is to use a password and a salt like a normal scheme plus other parameters, and you can think of the salt as an index into a large set of keys derived from the same password, and of course you don't need to hide the salt to operate.

Introduction:

Storing passwords is really hard mechanism to handle. Because storing passwords in a plaintext format will allow any attacker to get into users account immediately, and now a days developers tend to store passwords in 'hashed' format by injecting passwords into a hashing functions, and just storing the hash that come out of it, But also this technique is not safe from a security perspective and that's because there is an trivial attack and that's a 'rainbow table' attack and it's simply pre-calculated tables with the passwords on plaintext format mapped to its digests based on a specific algorithm and it can be found easily on the internet and it's so fast, So we are introducing a new function called "TOHA" and it's simply a function that will take a password and turn it to digest with specific length but this time this function will take number of arguments to work and it will make it hard and costly to an attacker to perform some attacks like the rainbow table attack, This function is a one-way function, the Idea behind this function is injecting the user password into a function with different number of rounds to make it hard to an attacker to preform brute-force attack, And that's because the time it will take the attacker to check every single candidate will increase, in other words the rate between every password guessing is really important, and that's simply because this function will slow up the password generation of the hashed passwords.

Related Work:

In this section we describe the related work of some of the current schemes of deriving keys from passwords and what is the strong and week points in a perspective of the features of TOHA function.

Bcrypt[]:

Bcrypt uses a block cipher (blowfish) to create the underlying compression function[1], it's used to create digests for password in slower mechanism to defend against the generic attacks, So this is the strong point in Bcrypt, and Bcrypt depends heavily on access to a table which is constantly altered through the algorithm execution, and this is very fast on PC, much less on GPU or FPGAs, where memory is shared and all cores compete for control of the internal memory bus, but in TOHA it's different in a sense that

TOHA is not dependent on any tables it relays on the data that come out of the processing mechanism, so if an adversary try to crack the round function he will have to do the calculation and then try to use the produced value that come out of the round function and then use it, so the states will be different every time in perspective of the round function results.

PBKDF2[]:

PBKDF2 is password based key derivation function that published by RSA laboratories in PKCS5[2] and its used to make the password generation mechanism really slow to defend against brute force attacks, PBKDF2 is based on HMAC Construction using HMAC-SHA1[2] and then invoke it many times as specified by the iteration parameters, While in TOHA it's different it's not about invoking the construction many times, in a sense that the pre-processing of the parameter need it to inject it to the round function, so the adversary can't distribute it easily on parallel design, when PBKDF2 can be disrepute on GPU that do parallel cracking, and in PBKDF2 useless processing of data involving XOR operation $\mathrm{SHA1}(\mathrm{P} \oplus \mathrm{ipad})$ and $\mathrm{SHA}-1(\mathrm{P} \oplus \mathrm{opad})$ [2] and then use such value for c times, In so doing , an attacker is able to avoid 50% of the operations involved in the key derivation process[3].

Algorithm Primitives:

Constant Parameters:

s = Salt || m = number of Iterations in the Initial step || k = Second step state
p = password || n = number of rounds || Q = Initial Vector
R = Initial step result || F = final result
x = SHA-256(p||p.length||s.length||s) = Key Generation

Overview:

TOHA-256 operates on 32-bit words and return 256-bit hash value, TOHA function will take some arguments to work, for starter the initialization vector that denoted by $\boldsymbol{Q}$ will be the same initialization value (IV) of SHA-256.

$$Q_0 = 0x6a09e667 \quad Q_1 = 0xbb67ae85 \quad Q_2 = 0x3c6ef372 \quad Q_3 = 0xa54ff53a$$
$$Q_4 = 0x510e527f \quad Q_5 = 0x9b05688c \quad Q_6 = 0x1f83d9ab \quad Q_7 = 0x5be0cd19$$

Variables:

The $\boldsymbol{x}$ constant will be the generated key that will enter the TOHA function in other words the user password will not be entered directly to the function it will be combined with a salt and then will be suited to be used in the function.

The $\boldsymbol{k}$ variable will hold the states that required to be used in the second step and this state will be generated randomly in the first step.

The $\boldsymbol{m}$ constant will be the number of iteration in the initial step.

The $\boldsymbol{n}$ constant will be the number of rounds in the second step.

The $\boldsymbol{R}$ variable will be the final result of the initial step and it will be entered directly to be the seed of the second step along with the $\boldsymbol{k}$ variable.

The $\boldsymbol{F}$ variable will be the final result that you can use in your application.

Operations:

$\oplus$: Bit-wise exclusive OR
|| : Concatenation
<<< : left-rotation (shift by one)

TOHA in memory:

The value we use in memory in this mode of operation is the array **K** that hold the states that required in the second step and it filled with data over and over again.

$$k[0] = H_1[\ll<];$$

$$for\ i\ from\ 1\ to\ n$$

$$k[0] = H_2(m * 2||x||R\hat{}k[<<<])^1, H_2(m * 2||k[y]||R\hat{}k[<<<])^2, \ldots., H_2(m * 2||k[y]||R\hat{}k[<<<])^{n-1}$$

Where x is the result from the first step and **k[y]** is the result of each round of the second step.

We have two type of the indexing functions $\emptyset()$[4]:

- Independent of the password and salt but depends on other public parameters that is available in the scheme, the memory addresses can be computed by the adversary, we assume that the cracking machine handle parallel memory access, so an adversary can fetch the results and implements a time space tradeoff, then he can compute the missing block, but then the round function calls will occupy the area to the β of the entire memory, where C(∝) will be the core needed to implement Fig 1.

$$\varepsilon(\propto) = \frac{1}{\propto + C(\propto)\beta}$$

- Dependent of the password, if an adversary got this data without the other parameters in this case $\emptyset(i) = H_1(p)^1, .., H_1(p)^{m-1}$, so he will be able to re-compute the missing data, the adversary has the primary information to get the results, but he will be missing one elements and that's time, if we think about the calls as tree of **A** the average depth **B**, then the total computation time will be multiplied by **B** Fig 2 .

$$\varepsilon(\propto) = \frac{1}{\propto + C(\propto)\beta B(\propto)}$$

Algorithm Specifications:

Initial Step:

In the initial step we will try to shuffle the bytes and make a diffusion layer to the key and generate a key and some other states to be used in the second step, the hash we use in this step Is SHA-256, and it works as follow, the key will be xored with the Q (initialization value) and the result will be concatenated with the m variable as a counter and then will be injected into the hash function, the result will be xored again with the key and concatenated again with m variable and injected to the hash function, after two steps the result will be injected into a shift Column function and that will shift the result one step to the left side and the result of the function will be the state **K** that will be used in the round function, after this the result will complete the cycle until we reach the end of the **m** variable without using the shift column function ever again Fig 1.

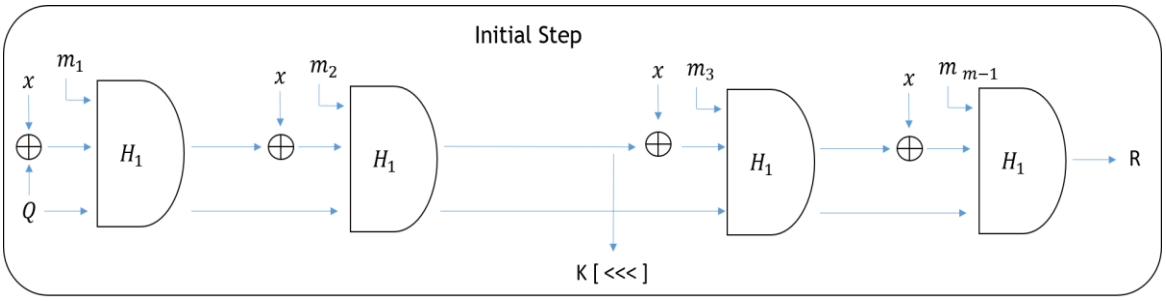

Fig 1 – Initial step

Second Step:

In the second step we will try to make the final result to be stored as the user password, this step will slow up the generation of the hashed value, the hash we use in this step is BLAKE [5]And it works as follow, the result generated from the previous step will be xored first with the **K** state and the result will be xored with the key but this result will be updating the **K** state to be used in the remaining steps, after this the 2 results will be concatenated with the m variable times 2, after every cycle of the round function the result will be injected into mix rows function and these steps will be done until we reach the end of **n** variable Fig 2.

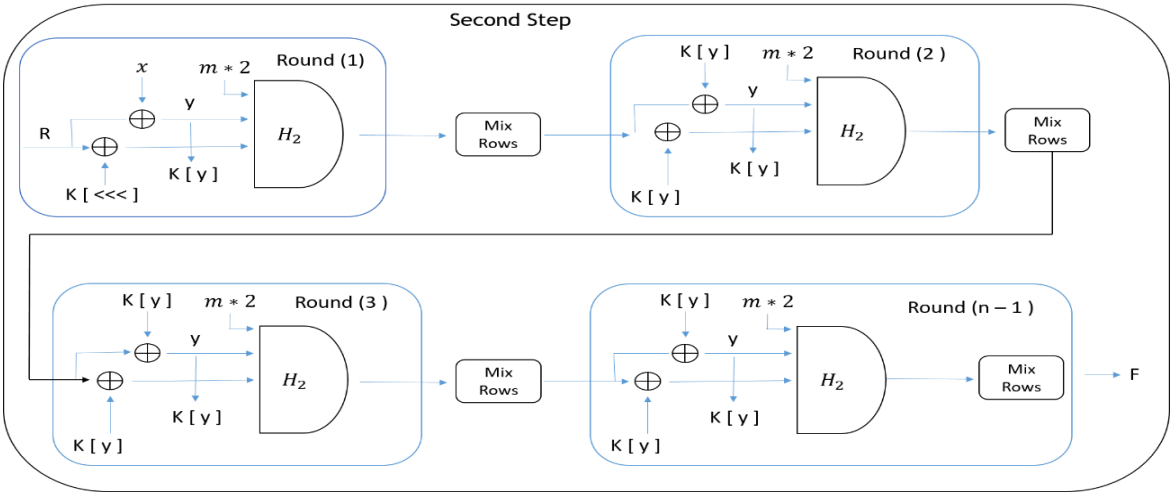

Fig 2 – Second Step (round function)

Salt Generation:

Salt is a pseudo random data it's used to give additional security for passwords schemes, and to be more specific to defend against generic attacks such dictionary or pre-computed rainbow tables, TOHA is using 128-bit random values, we used version 4 of UUID to achieve pseudo randomness, the number of random version 4 UUIDs which need to be generated in order to have 50% of probability of at least on collision is 2.71 quintillion computed as follows: $n \cong \frac{1}{2} + \sqrt{\frac{1}{4} + 2 \times \ln(2) \times 2^{122}} = 2.71 \times 10^{18}$.

Round Function:

1- $R \oplus k[<<<]$
2- $R \oplus x$
3- $k[y] \rightarrow R \oplus x$
4- $F((m \times 2) \,||\, R \oplus x \,||\, R \oplus k[y])$
5- $Mix\ Rows\ (output)$ $Fig\ 3$

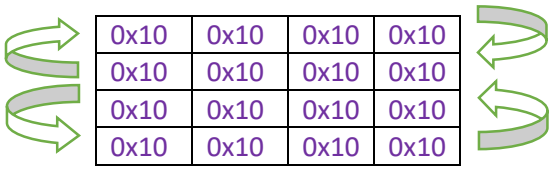

Fig 3 – Mix rows

Number of rounds:

The number of rounds are defined by the user, we gave the ability to change the parameters as per requirements and we will give the best parameters to use in the "Recommended parameter section".

Crypto Analysis:

General Security Definition:

Definition: *An attack on a hash function is a non-generic method of distinguishing the hash function from an ideal hash function* [6], the core function of TOHA is using hash function in its internal process. So it's important to define the ideal hash function, an ideal hash function is one that maps the set of actual key values to the table without any collisions. But that's just a theoretical approach.

Randomness, Preimage/Collision Resistance:

A preimage of a given hash value, H, is any message, M, such that Hash(M) = H, Preimage resistance describes the security guarantee that given a random hash value, an attacker will never find a preimage of that hash value [7]These three properties which define the security of hash algorithms, and we are trying to perform a large amount of computation to achieve randomness besides that the password is already hidden from the attacker and the number of states that been choose randomly to make it hard to do these kind of attacks.

Compute Time Cost:

TOHA is a sequential function so we are calling the hash algorithm sequentially, the attacker is forced to spend a comparable time computing the result of the hash function, TOHA is not relaying on XOR operations only because this can be distributed and parallel computation could be used to quick up the process, since XOR is not the only process the attacker is forced to calculate the result sequentially even on a custom ASIC.

Cycle Detection [11]:

Cycle detection it's an algorithmic step for finding sequence in the given values, let's assume infinite set $\{x_i = F(x_i - 1)\}\ i > 0\}$ where F is a function that maps a finite set N to itself if we used initial value x_0 in N F is expected to have collision $F(x) = F(x^{\sim})$ where $x \neq x^{\sim}$ we tried to use Floyd's two finger algorithm for detecting if there are any collision, the algorithm work as follows: you have two pointers one of them run at normal speed $(x \to f(x))$ and the other at double speed $(x \to f(f(x^{\sim}))$ until they collide. In TOHA we used the initial step to prepare the inputs and the states to ensure that there is no duplication and make sure that the values will not be repeated twice.

Side channel attacks:

Cache and Timing attacks [7]:

Depending on the platform, different execution times. This could lead to timing attacks or to cache timing attacks, either way we have two arrays one for holding the k state and the other for the results of the initial step, the k array will be overwritten many times in the second step and the R array will be injected as the seed for the second step therefore even if an attacker has a copy of the value he can't get around because the value it's not dependent on the password alone.

Slide attack [8]:

The attack relied on the fact that the cipher has identical sub keys in each round, typical approach to defend against slide attack is to introduce different state every round, and in TOHA we are providing different state in the round function, and if there is any slide change it will result a different output.

Length extension attack:

The length extension attack is an attack based on a vulnerability on hash functions, basically it will allow an attacker to append a value of his choice and generate a valid digest without the knowledge of the secret values that already been calculated $result = H(secret||value)$, and lets add m' where this value is the value that the attacker want to append $result' = H(secret||value)||m'$ and strangely enough $result = result'$ and this done by trying to get the calculated digest and trying to get the hash function end with the same state and the hash function will pick up where it left off and generate a new value as the new digest. This attack does not apply to TOHA for many reasons, you can't end up with the same state because of the input of the number of bits hashed so far to the compression function that $BLAKE$ [5] uses which simulates a specific output function for the last message block, beside

in TOHA construction password it's not the only input to the function it will require other arguments for that it's feasible to apply length extension attack for it. And mostly length extension attack PADDING that consist only from the bits defined by the padding rule of that hash function [9].

Results:

Let's assume we have this password: **toha-256**:

$$\$t0\$MzNmODFhNDBhNGU2NDRjNjkwNjFlYzM0Y2FhYWMzYzc = \$rtnySl6FMgc1N3/NiPDc/9jfmXKJ3O/3Af20kQH/mo0 =$$

its consist of 3 groups separated by **$**, first group it's just an indication for TOHA Function, second group is the salt, third group is the hashed key.

Using TOHA:

TOHA function can be used as a key derivation function (KDF) to derive a key from a password and its suitable to be used for generating 256-bit keys to be used for example in a symmetric encryption schemes, another use of TOHA Is to store passwords in a hashed format to prevent hackers to get the actual passwords from the databases, and prevent some generic attacks like rainbow tables and dictionary attacks, TOHA can be used also in generating session keys and verifying them from a master token.

Recommended Parameters:

We recommend using at least 6 characters password and if you used passphrase instead of normal password that will be much better, for the **m**,**n** variable we suggest using 2^{15} value and above for both and the more you increased the value TOHA will take time to generate your key and it will give you more security.

Implementation:

implementation code is available on the following website: almorabea.net/toha

Conclusion:

We presented TOHA as password hardened function that allow users to define the amount of memory and rounds according to their need and resources in the target platform, we tried to make it hard for an attacker to use parallel implementation to speed up the process of cracking the target password, also we tried to cover some attacks like side channel attacks and more… and in TOHA you can generate set of keys for a single password to be used in other purposes.

References:

[1] Niels Provos and David Mazières, A Future-Adaptable Password Scheme, Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 11,1999

[2] RSA Laboratories: PKCS #5 V2.1: Password Based Cryptography Standard (2012)

[3] On the weaknesses of PBKDF, Andrea Visconti??, Simone Bossi, Hany Ragab, and Alexandro Calò, International Association for Cryptologic Research "IACR"

[4] Alex Biryukov, Daniel Dinu,Dmitry Khovratovich, Fast and Tradeoff-Resilient Memory-Hard Functions for Cryptocurrencies and Password Hashing, International Association for Cryptologic Research "IACR"

[5] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan, "SHA -3 proposal BLAKE", version 1.3, December 16, 2010

[6] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno, Cryptography Engineering Design Principles and Practical Applications. Wiley (2010)

[7] Jean-Philippe Aumasson, Serious Cryptography, No Starch Press,2018

[8] Billy Bob Brumley and Nicola Tuveri, Remote Timing Attacks Are Still Practical, vol 6879. Springer, Berlin, Heidelberg (2011)

[9] Alex Biryukov, David Wagner. Slide Attacks. Fast Software Encryption – FSE'99, pp.245 - 259.

[10] Danilo Gligoroski, Length Extension Attack on Narrow-Pipe SHA-3 Candidates, Springer, Berlin, Heidelberg

[11] Antonie Joux, Algorithmic Cryptanalysis, CRC Press