

# ciphart

## memory-hard key derivation with easier measurable security

caveman<sup>1</sup>  
2021-01-11 02:10:59+00:00

*argon2*<sup>2</sup> is mostly nice, but trying to interpret its contribution to the protection against password brute-forcing attacks remains more difficult than it should be. this vagueness is a problem that is not limited to *argon2*, but also shared with every other key derivation function that i've known so far.

when one uses *argon2*, his derived key will surely have superior protection against password brute-forcing attacks, but by how much? to answer this, one would need to survey the industry that manufactures application-specific integrated circuits to obtain a map between *time* and *money*, in order to get an estimation on how much would it cost the adversary to discover the password in a given time window.

resolving this vagueness is not a mere luxury to have, but a necessity for maximising survival, because it hinders the process of studying the cost-value of memory-hard key derivation functions, which, effectively, increases the risk of having a false sense of security.

so i propose *ciphart* — a memory-hard key derivation function with a security contribution that is measured in a unit that i call *relative entropy bits*. this unit is measured objectively and is guaranteed to be true irrespective of whatever alien technology that the adversary might have.

**libciphart**<sup>3</sup> is a library that implements *ciphart* very closely to this paper, without much fluff. this should make integrating *ciphart* into other systems more convenient.

**ciphart**<sup>4</sup> is an application for encrypting and decrypting files that makes use of **libciphart**. this application is intended for use by end-users or scripts, henceforth it has some fluff to treat mankind with dignity.

## 1 ciphart

### 1.1 parameters

<i>enc</i>	encryption function.
<i>p</i>	password.
<i>s</i>	salt.
<i>L</i>	number of memory lanes for concurrency.
<i>M</i>	total memory in bytes.
<i>T</i>	number of tasks per lane segment.
<i>R</i>	number of rounds per task.
<i>B</i>	added security in <i>relative entropy bits</i> .
<i>K<sub>out</sub></i>	output key size in bytes.

<sup>1</sup>toraboracaveman[at]protonmail[dot]com

<sup>2</sup><https://github.com/P-H-C/phc-winner-argon2>

<sup>3</sup><https://github.com/Al-Caveman/libciphart>

<sup>4</sup><https://github.com/Al-Caveman/ciphart>

### 1.2 internal variables

$$C \leftarrow \begin{cases} 64 \text{ bytes} & \text{if enc is } xchacha20 \\ 16 \text{ bytes} & \text{if enc is } aes \\ \dots \end{cases}$$

this to reflect the block size of the encryption algorithm that's going to use *ciphart*'s generated key to encrypt data.

$$K_{in} \leftarrow \begin{cases} 32 \text{ bytes} & \text{if enc is } xchacha20 \\ 16 \text{ bytes} & \text{if enc is } aes-128 \\ \dots \end{cases}$$

this is the size of the encryption key that's used to solve *ciphart*'s tasks. this is different than the enc-independent *K<sub>out</sub>* which is possibly used by other encryption algorithms in later stages<sup>5</sup>.

$$\hat{T} \leftarrow T - (T \bmod 2) + 2. \text{ this is to ensure that it is in multiples of 2. why? because we need a buffer for storing the clear-text and another for storing the output cipher-text.}$$

$$\hat{M} \leftarrow M - (M \bmod C\hat{T}L) + C\hat{T}L. \text{ this is to ensure that it is in multiples of } C\hat{T}L. \text{ why? so that all segments are of equal lengths in order to simplify } ciphart's \text{ logic. e.g. it wouldn't be nice if the last segments were of unequal sizes.}$$

$$G \leftarrow \hat{M}C^{-1}\hat{T}^{-1}L^{-1}. \text{ total number of segments per lane.}$$

$$\hat{B} \leftarrow \max(\log_2(\hat{M}C^{-1}R), B). \text{ this is to ensure that } \hat{B} \text{ is large enough to have at least one pass over the } \hat{M}\text{-bytes memory.}$$

$$\hat{B} \leftarrow \log_2(2^{\hat{B}} - (2^{\hat{B}} \bmod L\hat{T}R) + L\hat{T}R). \text{ this is just to reflect the reality with } ciphart \text{ that segments must complete. i.e. when the user asks for } B \text{ relative entropy bits, he gets } \hat{B} \text{ instead, where } \hat{B} \geq B. \text{ more details on this later.}$$

$$m_i \leftarrow C\text{-bytes memory for } i^{th} \text{ task in the } \hat{M}\text{-bytes pad.}$$

$$n_l \leftarrow l\hat{T}R. \text{ nonce variable for } l^{th} \text{ lane with at least 64 bits.}$$

$$f \leftarrow 0. \text{ a flag indicating whether the } \hat{M}\text{-bytes pad is filled.}$$

### 1.3 output

$$k \leftarrow K\text{-bytes key with } \geq B \text{ relative entropy bits.}$$

### 1.4 steps

shown in algorithm 1.

## 2 parallelism

since iterations of the loop in line 3 in algorithm 1 are fully independent of one other, they can quite happily utilise *L* cpu cores, specially when segment sizes, *T*, are larger.

other lines are not easily independent, so i didn't even bother to try to parallelise them. specially since this is

<sup>5</sup>at the expense of losing the meaning of *relative entropy bits*.

---

**algorithm 1:** ciphart version 6

---

```
1 while 1 do
2   for  $g = 0, 1, \dots, G - 1$  do
3     for  $l = 0, 1, \dots, L - 1$  do
4       for  $t = 0, 1, \dots, T - 1$  do
5         for  $r = 0, 1, \dots, R - 1$  do
6            $i \leftarrow gT + t$ ;
7           if  $t = 0$  then
8              $j \leftarrow i + T - 1$ ;
9           else if  $t = T - 1$  then
10             $j \leftarrow i + 1 - T$ ;
11          else
12             $j \leftarrow i + 1$ ;
13           $m_j \leftarrow \text{enc}(m_i, n_l, k)$ ;
14           $n_l \leftarrow n_l + 1$ ;
15           $k \leftarrow f(m_j, p, l, s, t)$ ;
16        if  $f$  and  $\log_2(n_l L) \geq B$  then
17          go to line 19;
18       $f \leftarrow 1$ ;
19 while 1 do
20   for  $l = 1, 2, \dots, L$  do
21     if  $\text{len}(k) \geq K$  then return  $k[0 : K]$ ;
22      $n \leftarrow n + 1$ ;
23      $k \leftarrow k \parallel \text{enc}(m_{l,S,T}[1], n, k)$ ;
```

---

- 3 memory-hardness
- 4 security interpretation
- 5 comparison
- 6 summary

not a problem, since the cpu-heavy part is in the easily-parallelisable part.

with *argon2*, if one wants to increase the cpu load without increasing memory pad's use, one can increase the number of passes over the pad. this feature is supported by *ciphart* via the *relative entropy bits* parameter  $B$ .

but, simply increasing number of passes on the pad may not be the best option for all cases. e.g. what if someone has a small pad, and small segments? in such case, a higher percentage of the cpu will be wasted in the non-parallelisable steps, which is a waste.

this is why *ciphart* has an additional parameter  $R$ . this parameter can allow to increase the load on the cpu without even requiring to go through the non-parallelisable steps. *argon2* lacks this parameter. this is not the main reason *ciphart* was made, but it's one of the incremental improvements.

one could philosophically argue that *argon2* has the  $R$  parameter, except that it always assumes that  $R = 1$ . i don't agree with this assumption, which is why i made *ciphart* to allows the user to set  $R$  more flexibly.