ciphart

memory-hard key derivation with easier measurable security

caveman[1]

January 10, 2021

argon2[2] 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*. the centre of my thesis is that this part is not nice, because i found that life can be easier.

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`[3] 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`[4] 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

enc	encryption function.
p	password.
s	salt.
L	number of memory lanes for concurrency.
M	total memory in bytes.
T	number of tasks per lane segment.
R	number of rounds per task.
B	added security in *relative entropy bits*.
K_{out}	output key size in bytes.

[1] toraboracaveman [at] protonmail [dot] com
[2] `https://github.com/P-H-C/phc-winner-argon2`
[3] `https://github.com/Al-Caveman/libciphart`
[4] `https://github.com/Al-Caveman/ciphart`

1.2 internal variables

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

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

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

this is the size of the encryption key that's used to solve *ciphart*'s tasks.

$\hat{T}$ $\leftarrow T - (T \bmod 2) + 2$. 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.

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

G $\leftarrow \hat{M}/C/\hat{T}/L$; total number of segments per lane.

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

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

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

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

f $\leftarrow$ **false**; a flag indicating whether the $\hat{M}$-bytes pad is filled.

1.3 output

k K-bytes key with $\geq B$ *relative entropy bits*.

1.4 steps

shown in algorithm 1.

algorithm 1: ciphart version 6

```
1  while true do
2      for g = 0, 1, ..., G − 1 do
3          for l = 0, 1, ..., L − 1 do
4              for t = 0, 1, ..., T − 1 do
5                  for r = 0, 1, ..., R − 1 do
6                      i ← gT + t;
7                      if t = 0 then
8                          j ← i + T − 1;
9                      else if t = T − 1 then
10                         j ← i + 1 − T;
11                     else
12                         j ← i + 1;
13                     m_j ← enc(m_i, n_l, k);
14                     n_l ← n_l + 1;
15                     k ← f(m_j, p, l, s, t);
16         if f = true and log_2(n_1 L) ≥ B then
17             go to line 19;
18     f ← true;
19 while true do
20     for l = 1, 2, ..., L do
21         if len(k) ≥ K then return k[0 : K];
22         n ← n + 1;
23         k ← k ∥ enc(m_{l,S,T}[1], n, k);
```

2 parallelism

3 memory-hardness

4 security interpretation

5 comparison

6 summary