ciphart

memory-hard key derivation
with easier measurable security

caveman[1]

2021-01-22 13:43:24+00:00

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 (asics) 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.

while the approach of surveying the asics industry is not wrong, it is largely subjective, with expensive housekeeping, and practically leads the user to rely on vague foundations to build his security on. this vagueness is not nice, and it would be better if we had an objective measure to quantify the security of our memory-hard key derivation functions.

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]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`

content

1 ciphart

1.1 parameters

enc	encryption function.
hash	hashing function.
p	password.
s	salt.
M	total memory in bytes.
L	number of memory lanes for concurrency.
T	number of tasks per lane segment.
B	minimum quantity of increased protection against password brute-forcing attacks in the unit of *relative entropy bits.*
K	output key size in bytes.

1.2 internal variables

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

this to reflect the block size of the encryption algorithm that implements enc.

$$V \leftarrow \begin{cases} 32 \text{ bytes} & \text{if enc is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if enc is } \textit{aes-128} \\ 32 \text{ bytes} & \text{if enc is } \textit{aes-256} \\ \dots \end{cases}$$

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

$\hat{T}$ $\leftarrow \max(\lceil VC^{-1}\rceil, T)$. this is to ensure that we have enough encrypted bytes for new keys.

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

[5]at the expense of losing the meaning of *relative entropy bits.*

$\hat{M}$	$\leftarrow M - (M \bmod C\hat{T}L) + C\hat{T}L$. 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	$\leftarrow \hat{M}C^{-1}\hat{T}^{-1}L^{-1}$. total number of segments per lane.
$\hat{B}$	$\leftarrow \max(\hat{M}C^{-1}, 2^B)$. this is to ensure that $\hat{B}$ is large enough to have at least one pass over the $\hat{M}$-bytes memory.
$\hat{B}$	$\leftarrow \hat{B} - (\hat{B} \bmod L\hat{T}) + L\hat{T}$. this is to reflect the reality that, with *ciphart*, segments must complete. i.e. when the user asks for B *relative entropy bits*, he gets $\log_2 \hat{B}$ bits instead, where $\log_2 \hat{B} \geq B$.
m_i	C-bytes memory for i^{th} task in the $\hat{M}$-bytes pad.
n_l	$\leftarrow lG\hat{T}$. nonce variable for l^{th} lane with at least 64 bits.
f	$\leftarrow 0$. a flag indicating whether the $\hat{M}$-bytes pad is filled.
v	$\leftarrow *\text{hash}(p \parallel s, V)$. a pointer to the first byte where V-bytes key is stored.

1.3 output

k	K-bytes key with an increased protection against brute-forcing attacks by $\hat{B}$ *relative entropy bits*.

1.4 steps

steps of *ciphart* is shown in algorithm 1. this corresponds to *argon2d*. adding a *ciphart-i* variant is a trivial matter, i just didn't do it yet because my threat model currently doesn't benefit from a password independent variant.

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.

3 memory-hardness

Proof. algorithm 1 is just a variation of *argon2d*, except that it uses an encryption function, enc, instead of a hashing functionn. so if *argon2d* is memory-hard, then so is *ciphart*. □

4 security interpretation

5 comparison

6 summary

algorithm 1: ciphart

```
1  while 1 do
2      for $g = 0, 1, \ldots, G-1$ do
3          for $l = 0, 1, \ldots, L-1$ do
4              for $t = 0, 1, \ldots, T-1$ do
5                  $i \leftarrow gLT + lT + t$;
6                  if $t < T-1$ then
7                      $j \leftarrow i + 1$;
8                  else if $t = T-1$ then
9                      $j \leftarrow i - T + 1$;
10                 $m_j \leftarrow \text{enc}(m_i, n_l, v)$;
11                 $n_l \leftarrow n_l + 1$;
12                 if $f = 0$ then
13                     $v \leftarrow m_j \bmod (gLTC + tC - V)$;
14                     if $v \geq gLTC - V$ then
15                         $v \leftarrow v + lTC$;
16                 else
17                     $v \leftarrow m_j \bmod (GLTC - LTC + tC - V)$;
18                     if $v \geq gLTC + tC - V$ then
19                         $v \leftarrow v + LTC$;
20                     else if $v \geq gLTC - V$ then
21                         $v \leftarrow v + lTC$;
22         if $n_0 L = \hat{B}$ then
23             $g_{\text{last}} \leftarrow g$;
24             go to line 26;
25     $f \leftarrow 1$;
26 $i \leftarrow g_{\text{last}}LT + T - 1$;
27 $k \leftarrow \text{hash}(m_i \parallel m_{i+T} \parallel m_{i+2T} \parallel \ldots \parallel m_{i+LT}, K)$;
28 $t \leftarrow T - 1$;
29 for $l = 0, 1, \ldots, L-1$ do
30     $m_{i+t} \leftarrow \text{enc}(m_{i+t-1}, n_0, v)$;
31     $n_0 \leftarrow n_0 + 1$;
32     $v \leftarrow m_{i+t-1} \bmod (GLTC - LC + lC - V)$;
33 return $k$
```