ciphart

memory-hard key derivation with easier measurable security

caveman[1]

2021-01-11 22:15:57+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 ciphart

1.1 parameters

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

1.2 internal variables

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

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

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

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

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

$\hat{M}$ $\leftarrow M - (M \bmod C\hat{T}L) + C\hat{T}L$. 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.

$\hat{B}$ $\leftarrow \max(\log_2(\hat{M}C^{-1}R), B)$. 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 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 $\leftarrow gl\hat{T}R$. 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.

[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`

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

1.3 output

k K_{out}-bytes key with $\geq B$ *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, but i didn't do it *yet* because my threat model currently doesn't benefit from as such.

algorithm 1: ciphart

```
1  =1while 1 do
2      if $f = 0$ then $\hat{R} \leftarrow 1$ else $\hat{R} \leftarrow R$;
3      for $g = 0, 1, \ldots, G-1$ do
4          for $l = 0, 1, \ldots, L-1$ do
5              for $t = 0, 1, \ldots, T-1$ do
6                  $i \leftarrow gLT + t$;
7                  if $t < T-1$ then
8                      $j \leftarrow i + 1$;
9                  else if $t = T-1$ then
10                     $j \leftarrow i - T + 1$;
11                 for $r = 0, 1, \ldots, \hat{R}-1$ do
12                     $m_j \leftarrow \text{enc}(m_i, n_l, k)$;
13                     $\hat{i} \leftarrow i$;
14                     $i \leftarrow j$;
15                     $j \leftarrow \hat{i}$;
16                     $n_l \leftarrow n_l + 1$;
17                     if $f = 0$ then
18                         $v \leftarrow m_j \bmod (gLTR + t)$;
19                         if $v \geq gLTR$ then
20                             $v \leftarrow v + lT$;
21                     else
22                         $v \leftarrow m_j \bmod (GLTR - LTR + t)$;
23                         if $v \geq gLTR$ then
24                             $v \leftarrow v + LTR$;
25                     $k \leftarrow m_v[0 : K_{\text{in}}]$;
26         if $f$ and $\log_2(n_1 L) \geq B$ then
27             go to line 29;
28     $f \leftarrow 1$;
29 while 1 do
30     for $l = 1, 2, \ldots, L$ do
31         if $\text{len}(k) \geq K_{out}$ then return $k[0 : K_{\text{out}}]$;
32         $n \leftarrow n + 1$;
33         $k \leftarrow k \,\|\, \text{enc}(m_{l,S,T}[1], n, k)$;
```

2 parallelism

since iterations of the loop in line 4 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 not a problem, since the cpu-heavy part is in the easily-parallelise-able 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-parallelise-able 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-parallelise-able steps. *argon2* lacks this parameter. this is not the main reason *ciphart* was made, but it's one of the incremental improvements.

philosophically, i think that *argon2* does have the R parameter, except that it assumes that $R = 1$, and does not allow the user to change it. i don't agree with this assumption, which is why i made *ciphart* to allow the user to set R more freely.

however, i do agree that $R = 1$ can be a sensible *default* for most parameter and system combinations, but this is not a reason to force $R = 1$ for all other parameters or systems.

3 memory-hardness

4 security interpretation

5 comparison

6 summary