ciphart

memory-hard key derivation with easier measurable security

caveman[1]

2021-01-29 18:13:28+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.

paper's layout

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.
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

`hash` hashing function.

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

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

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

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.

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

[1]mail: 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*.

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

N $\leftarrow 0$. actual number of times `enc` is called, where $\hat{N} \geq 2^B$.

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 $\leftarrow *\texttt{hash}(p \parallel s, V)$. 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 $\log_2 N$ *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.

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         $m_j \leftarrow \texttt{enc}(m_i, n_l, v)$;
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           $v \leftarrow m_j \bmod (\hat{M} - LTC + tC - V)$;
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     $N \leftarrow N + LT$;
23   **if** $N \geq 2^B$ **then**
24       $g_{\text{last}} \leftarrow g$;
25   **go to** line 27;
26   $f \leftarrow 1$;
27 $i \leftarrow g_{\text{last}}LT$;
28 $k \leftarrow \texttt{hash}(m_{i+0T} \| m_{i+1T} \| \ldots \| m_{i+(L-1)T}, K)$;
29 **return** k

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 comparison

4.1 raw

5 security interpretation

5.1 raw

let's say that we used block encryption function `enc` and a key $v \leftarrow \texttt{hash}(p\|s, V)$ to encrypt some clear-text into a sequence of C-byte cipher-text blocks $m_0, m_1, \ldots$. let's say that the adversary got those $m_0, m_1, \ldots$.

adversary's goal is to decrypt those cipher-text blocks back into the original clear-text. so what options does he have?

- *option 1:* brute-force the V-bytes key space. in order to get a probability of 1 of finding the key v, the adversary would need to evaluate 2^{8V} many keys[6]. 2^{8V} is too large of a number, and i think maybe the adversary would be fossilised

  this works by having the adversary repeatedly decrypting m_0 with `enc`, each time using a new key among

  - $\hat{v}_0 \leftarrow$ `0x00...0`,
  - $\hat{v}_1 \leftarrow$ `0x00...1`,
  - $\vdots$
  - $\hat{v}_{8V-1} \leftarrow$ `0xff...f`,

  until the adversary finds a key that manages to decrypt m_0.

[6] assuming that each byte is 8 bits.

the adversary could be extremely lucky and have v_0 manage to decrypt m_0, hence needing to call `enc` only once.

or he might be extremely unlucky and need to keep trying until $v_{2^{8V}-1}$ manages to do it, hence needing to call `enc` for 2^{8V} many times.

usually it's sometime in between. asymptotically on on average, the adversary would need to call `enc` for $2^{8V}/2$ many times.

but in order to guarantee finding v, the brute-forcing process would need to run 2^{8V} many evaluations, hence calling `enc` for 2^{8V} many times.

- *option 2:* brute-force the $H(p)$ bits password space. where $H(p)$ is the amount of entropy bits in p. in order to get a probability of 1 of finding the password p, the adversary would need to evaluate $2^{H(p)}$ many keys[7].

  this works by having the adversary repeatedly decrypting m_0 with `enc`, each time using a new key among:

  - $\hat{v}_0 \leftarrow \texttt{hash}(\hat{p}_0 \| s, V)$,
  - $\hat{v}_1 \leftarrow \texttt{hash}(\hat{p}_1 \| s, V)$,
  - $\vdots$
  - $\hat{v}_{2^{H(p)}-1} \leftarrow \texttt{hash}(\hat{p}_{2^{H(p)}-1} \| s, V)$,

  until the adversary finds a key that manages to decrypt m_0.

  in order to guarantee finding v, the brute-forcing process would need to call `enc` for $2^{H(p)}$ many times.

unless the adversary is an idiot, he will try to attack the weakest element in the chain: *option 2*, since $H(p) < 8V$ is almost always true for any p that is memorable by humans.

knowing the strategy above, we'd be very dumb to choose a password p with more entropy than $8V$ bits. because we would end up needlessly memorising a password that is too long. why? because if $H(p) > 8V$, the weakest element in the chain becomes *option 1* instead, and the attacker wouldn't bother with passwords, hence any extra password memorisation effort was pointless.

security interpretation: your protection against password brute-forcing attacks is $\min(H(p), 8V)$ entropy bits.

5.2 with *argon2*

adversary's options are:

- *option 1:* identical to the *raw* case.
- *option 3:* this is similar to *option 2* except for evaluating keys from:

  - $\hat{v}_0 \leftarrow \texttt{argon2}(\hat{p}_0, \ldots)$,
  - $\hat{v}_1 \leftarrow \texttt{argon2}(\hat{p}_1, \ldots)$,
  - $\vdots$
  - $\hat{v}_{2^{H(p)}-1} \leftarrow \texttt{argon2}(\hat{p}_{2^{H(p)}-1}, \ldots)$,

  where `enc` is the same function that's used to encrypt cipher-texts $m_0, m_1, \ldots$ and $K = V$.

5.3 with *ciphart*

adversary's options are:

- *option 1:* identical to the case without *ciphart*.
- *option 3:* this is similar to *option 2* except for evaluating keys from:

  - $\hat{v}_0 \leftarrow \texttt{ciphart}(\texttt{enc}, \hat{p}_0, s, M, L, T, B, K)$,
  - $\hat{v}_1 \leftarrow \texttt{ciphart}(\texttt{enc}, \hat{p}_1, s, M, L, T, B, K)$,
  - $\vdots$
  - $\hat{v}_{2^{H(p)}-1} \leftarrow \texttt{ciphart}(\texttt{enc}, \hat{p}_{2^{H(p)}-1}, s, M, L, T, B, K)$,

  where `enc` is the same function that's used to encrypt cipher-texts $m_0, m_1, \ldots$ and $K = V$.

why should it be the same encryption function and $K = V$? because it's the case where *ciphart* has a guaranteed interpretation as i show later on. *ciphart* may also have a guaranteed interpretation with different encryption functions and with $K \neq V$, but i don't know it yet. so let's stick with identical encryption functions and $K = V$ for now.

each time the function `ciphart` is called, the encryption function `enc` is called N many times, where $N \geq 2^B$. as per algorithm 1, there is no way for the adversary to cheat by reducing N for as long as `enc` is not broken. the guarantee that the adversary cannot reduce N is cryptographic, and is totally independent of the implementation of his brute-forcing apparatus.

so, when `ciphart` is called for $2^{H(p)}$ many times, it *necessarily* has to result in calling `enc` for $N2^{H(p)} = 2^{H(p)+\log_2 N}$ many times.

meaning, using `ciphart` with p would have the *same* computational effect to the case of using just `hash` but with a more complex password $\hat{p}$, such that:

$$H(\hat{p}) = H(p) + \log_2 N \tag{1}$$

also, since the encryption algorithm that's used by `ciphart` is the same as the one that's used to encrypt cipher-text blocks $m_0, m_1, \ldots$, we know that if it costs the adversary c-many units of money for a single call to `enc` with strategy *option 1*, it will necessarily have to cost him cN units of money with strategy *option 3*, because `ciphart` in *option 3* is using the same `enc` that's used in *option 1*.

this way, it doesn't matter to us what kind of alien technology that adversary has: if one call to `enc` costs him c

[7] the adversary does not know p, obviously, but he knows the process that the user used to generate p, henceforth he knows $H(p)$.

units of money, then using `ciphart` will make his calls to `enc` increase N fold.

security interpretation: your protection against password brute-forcing attacks is $\geq \min(H(p) + \log_2 N, 8V)$ *relative entropy bits*.

6 summary