ciphart

memory-hard key derivation with easier measurable security

caveman[1]

2021-02-08 21:08:52+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 *caveman's 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 shannon's entropy

we've got password p with $H(p)$ many shannon's entropy bits worth of information in it. so what does this mean?

fundamentally, it means that, on average, we'd need to ask $H(p)$ many perfect binary questions[5] in order to fully resolve all ambiguities about p; i.e. to fully get every bit of p.

but people use it to do less orthodox things, such as quantifying the amount of security p has against, say, brute-forcing attacks.

say that we've got a 256 bit key $k \leftarrow \texttt{hash}(p\|s, 256)$, derived from password p, where s is a salt. if an attacker has a cipher-text block $m_0 \leftarrow \texttt{enc}(\text{“goodbye, world”}, 0, k)$. attacker's goal is to find the key that encrypted the sequence $m_0, m_1, \ldots$. then the attacker has the following options:

1. try to decrypt m_0, each time with a new key, until it results in “goodbye, world” in order to conclude that k is found. this will require him to perform 2^{256} decryption calls in order to guarantee finding the key. or, $2^{256}/2$ on average. both of which are insanely slow and would take many centuries to complete. my laptop would take 4.28×10^{58} many centuries in order to just in increment a mere counter for 2^{256} times. so this is not practical.

2. if `enc` is broken enough by cryptanalysis, then using approach in step item 1 could become feasible. but only an idiot would use a broken `enc`, and such people need a different type of help than what this paper tries to offer. so this option is off-topic, and i won't discuss it any further.

3. explore key candidates by trying different passwords. while each step is more expensive as it also requires calling `hash`, the total number of iterations is $2^{H(p)}$,

[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] one which, if answered, and on average, gets the search space reduced in half.

which is usually small enough to make this strategy practically viable for the adversary to bet on. this paper is about enhancing our security against this option.

the amount of security that we have against brute-forcing options is the cost of brute-forcing. the more expensive the cost is, the harder it is on the adversary, and hence the more security we have. with the option in item 3 our cost is[6]:

$$2^{H(p)}\Big(\text{cost}(\texttt{hash}) + \text{cost}(\texttt{enc})\Big) \tag{1}$$

one may try to empirically estimate cost(`hash`) and cost(`enc`) in (1), by analysing the industry of asics in order to obtain a mapping between the total amount of money that's needed in order to identify p within a given time frame[7].

but this simple 1 to 1 mapping between cost and entropy is broken when expensive key derivation functions is used. say that the key k was obtained by recursively calling the `hash` function N many times. i.e. $\texttt{hash}(\texttt{hash}(\dots\texttt{hash}(p\|s, 256), \dots, 256), 256)$. this way

2 caveman's entropy

3 ciphart

3.1 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 *caveman's entropy bits*.
K	output key size in bytes.

[6] where cost(`hash`) is the cost of a single call to `hash`, and cost(`enc`) is that of `enc`.

[7] the *scrypt* paper contains such an attempt.

3.2 internal variables

`hash`	hashing function.
C	$\leftarrow \begin{cases} \text{64 bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ \text{16 bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes} \\ \dots \end{cases}$ this to reflect the block size of the encryption algorithm that implements `enc`.
V	$\leftarrow \begin{cases} \text{32 bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ \text{16 bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-128} \\ \text{32 bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-256} \\ \dots \end{cases}$ 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[8].
$\hat{T}$	$\leftarrow \max(\lceil VC^{-1}\rceil, T)$. 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.
$\hat{M}$	$\leftarrow M - (M \bmod C\hat{T}L) + C\hat{T}L$. 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.

3.3 output

k	K-bytes key.
$\hat{B}$	actual quantity of increased security against password brute-forcing attacks in the unit of *caveman's entropy bits*, where $\hat{B} \geq B$.

3.4 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.

[8] at the expense of losing the meaning of *caveman's entropy bits*.

algorithm 1: ciphart

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         $i \leftarrow gLT + lT + t$;
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 $k \leftarrow \texttt{hash}(m_{i+0T} \| m_{i+1T} \| \dots \| m_{i+(L-1)T}, K)$;
29 $\hat{B} \leftarrow log_2 N$;
30 **return** k, $\hat{B}$

4 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.

5 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*. □

6 security interpretation

note 1. *i assume that the decryption part of the encryption algorithm* `enc` *costs the same as the encryption. this is true for algorithms such as* xchacha20. *and in cases where it is not true, such as with aes,* ciphart *can simply encrypt using the decryption function. this way we guarantee that the cost are identical between* ciphart*'s encryption, and the cipher-text decryption that the adversary does to test a given key.*

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 $m_0, m_1, \dots$. let's say that the adversary got those $m_0, m_1, \dots$.

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

6.1 key brute-forcing

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[9].

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`,
- $\vdots$
- $\hat{v}_{2^{8V}-1} \leftarrow$ `0xff...f`,

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

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 n 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.

that said, the adversary would be fossilised long before his application completes. e.g. since $8V = 256$ is common for ciphers nowadays, on average while considering the lucky and the unlucky cases, it would take my laptop 4.28×10^{58} centuries to just increment a counter for 2^{256} many times. so if the adversary's best hope is to brute-force keys, our system has reached maximum security.

security interpretation 1. *your protection against key brute-forcing attacks with key brute-forcing is* 2^{8V} cost(`enc`), *where* cost(`enc`) *is the cost of executing* `enc` *a single time.*

this is usually called $8V$ entropy bits. *but for the purpose of helping later sections, i think it's better to call it* $8V$ entropy bits from the viewing angle of `enc`, *or, for short:*

$$8V \text{ HENC}$$

[9]assuming that each byte is 8 bits.

definition 1. HENC *is entropy bits from the viewing angle of* `enc`*.*

HENC is specific only to `enc`'s algorithm, so must hold with whatever alien technology that the adversary may have, for as long as `enc`, as an algorithm, has no cryptanalysis. if there is any cryptanalysis, we'll have to subtract bits from $8V$, e.g. $8V - z$ HENC, where z is number of reduced bits due to cryptanalysis.

6.2 normal password brute-forcing

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[10].

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.

security interpretation 2. *your protection against password brute-forcing attacks with normal hashed passwords is* $2^{H(p)}\,\mathrm{cost}(\texttt{hash}) + 2^{H(p)}\,\mathrm{cost}(\texttt{enc})$*. the latter* `enc` *calls are due to trying to decrypt* m_0 *at every attempt.*

this is usually called $H(p)$ entropy bits*. but for the purpose of this paper, i think it's better to be more specific. from the viewing angle of* `enc`*, we get the entropy:*

$$H(p)\ \text{HENC}$$

and from the viewing angle of `hash`*, we get the entropy:*

$$H(p)\ \text{HHASH}$$

this may seem silly as it is too obvious, but i think it helps me to communicate my thoughts in later sections.

definition 2. HHASH *is entropy bits from the viewing angle of* `hash`*.*

6.3 with *argon2*

adversary evaluates keys from:

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

for every *argon2* call, `hash` is called for N many times if there is M bytes of memory. so, from the viewing angle of `hash`, we get:

- $\hat{v}_0 \leftarrow \texttt{hash}(\hat{p}_0 \| s_0, V)$,
- $\hat{v}_1 \leftarrow \texttt{hash}(\hat{p}_1 \| s_1, V)$,
- $\vdots$
- $\hat{v}_{N2^{H(p)}-1} \leftarrow \texttt{hash}(\hat{p}_{N2^{H(p)}-1} \| s_{N2^{H(p)}-1}, V)$,

if an adversary lacks M bytes, he can still compute $\texttt{argon2}(p, N, M, \ldots)$, but at the expense of needing exponentially more cpu time as his memory is linearly reduced.

security interpretation 3. *your protection against password brute-forcing attacks under the* argon2 *protection is:*

$$\begin{aligned} & N2^{H(p)}\,\mathrm{cost}(\texttt{hash}) \qquad + 2^{H(p)}\,\mathrm{cost}(\texttt{enc}) \\ = & \ 2^{H(p)+\log_2 N}\,\mathrm{cost}(\texttt{hash}) + 2^{H(p)}\,\mathrm{cost}(\texttt{enc}) \end{aligned}$$

from the viewing angle of `enc` *we get the entropy:*

$$H(p)\ \text{HENC}$$

while from the viewing angle of `hash` *we get the entropy:*

$$H(p) + \log_2 N\ \text{HHASH}$$

so which one to pick? i think people so far just pick $H(p)$ HENC to reflect password's entropy, and seem to not pick $H(p) + \log_2 N$ HHASH as they don't seem to consider it entropy. but i have two disagreements with people:

- i think not accepting that $H(p) + \log_2 N$ HHASH is entropy is needlessly limiting. because i think $H(p) + \log_2 N$ HHASH is entropy as much as $H(p)$ HENC is entropy; it's just that they are measured from different viewing angles: former is measured from the `hash` viewing angle, while the latter is measured from the `enc` viewing angle.

  i don't see any reason why any of them is more true than the other. i think that both of them are entropies, but of different units.

- why do people only pick either one of them? it's technically false my view. in my view truth is: we're just dealing with two entropies measured in different units. so i think truth is that we have the following number of entropy bits:

  $$\begin{aligned} & H(p) && \text{HENC} \\ + & H(p) + \log_2 N && \text{HHASH} \end{aligned}$$

  which obviously looks a bit ugly, since we cannot sum them due to the terms having different units, which also gives our brain a hard time to get a feeling of what that even means.

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

so what's the solution here to this ugliness? should we ignore $H(p)$ HENC $+H(p)+\log_2 N$ HHASH as an entropy measure that quantifies the security of our protection against password brute-forcing, as people currently do, and measure it only in terms of the computational cost by surveying the industry of asics in order to find a map between time and money?

my answer to the questions above is:

- no. the right approach is to just admit that *argon2*'s approach is dragging us into the situation where we end up with two entropies measured in different units.
- *argon2*'s security contribution is measurable as entropy, except that it is ugly since it is made of two entropies in distinct units. if we ignore this, we won't solve the problem, but end up stashing the dirts under the carpet.
- of course, we are always free to also survey the industry of asics to derive time-money maps, but this doesn't have to be our only approach to quantify our security gain.

6.4 with *ciphart*

adversary evaluates keys from:

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

mostly similar to `argon2`. differences related to this section is that `ciphart` calls `enc` instead of `hash`, and specifies B instead of N, where $B \approx \log_2 N$. similar exponential time penalty applies with memory less than M.

security interpretation 4. *your protection against password brute-forcing attacks under the* ciphart *protection approach is:*

$$\begin{aligned}
&\left(2^{H(p)} + 2^{\hat{B}}\right) \text{cost}(\texttt{enc}) + 2^{H(p)} \text{cost}(\texttt{enc}) \\
=& \left(2^{H(p)} + 2^{\hat{B}} + 2^{H(p)}\right) \text{cost}(\texttt{enc}) \\
=& \left(2 \times 2^{H(p)} + 2^{\hat{B}}\right) \text{cost}(\texttt{enc}) \\
=& \left(2^{H(p)+\log_2 2} + 2^{\hat{B}}\right) \text{cost}(\texttt{enc}) \\
=& \left(2^{H(p)+1} + 2^{\hat{B}}\right) \text{cost}(\texttt{enc})
\end{aligned}$$

from the viewing angle of `enc` *we get the entropy:*

$$H(p) + 1 + \hat{B} \text{ HENC}$$

and there is no other viewing angle than `enc` since only `enc` is used! as a result our brain can easily interpret it.

plus, if we wish to study the industry of asics to obtain time-money maps, our job will be much easier as we can simply look at the cost of asics that are already implemented for `enc`[11].

7 summary

[11] e.g. if `enc` is a popular algorithm, such as aes-256, then we can get more specific data from manufacturers, ultimately giving us a more accurate time-money maps. but when `enc` is not popular, we may need to do rougher calculations based on the expected asics area as done in the *scrypt* paper.