ciphart

memory-harder key derivation with easier measurable security

caveman[1]

2021-02-15 04:59:25+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 background

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 $8V$ bit key $k \leftarrow \texttt{hash}(p \| s, 8V)$, derived from password p, where s is a salt. say that the attacker has s and k but wants to figure out p. in this case, he will need to brute-force the password space in order to find p that gives k. his cost is:

$$2^{H(p)} \Big(\texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k) \Big) \tag{1}$$

definition 1. *the security of a system is the cost of the cheapest method that can break it.*

one way to estimate `cost` is to survey the asics industry. by surveying the asics industry to get an idea how much money it costs to get a given key, or password, space brute-forced within a target time frame[6]. this has an expensive housekeeping and is usually not possible to get any guarantees as we don't know about state-of-art manufacturing secrets that adversaries may have. the *scrypt* paper has an example of such attempt.

another way is to ignore anything that has no cryptographic guarantee. so, in (1), cryptography guarantees[7] that $2^{H(p)}$ many `hash` calls are performed and that many equality tests. the `hash` call needs to be done once, so let's give it a unit of time 1. the equality test also needs to be

[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.
[6]see the *scrypt* paper for an example.
[7]statistically by confidence earned through peer review and attempts to break encryption algorithms.

called once, but since since it's so cheap it's easier to just assume that its cost is free. this way (1) becomes just:

$$2^{H(p)}(1+0) = 2^{H(p)} \tag{2}$$

further, for convenience, it seems that people report it in the $\log_2$ scale. i.e. $\log_2 2^{H(p)} = H(p)$. i think this is why people use password entropy as a measure of its security. not because it is the quantity of security, but rather because its the quantity of *simplified* security.

2 caveman's entropy

2.1 recursive `hash`

if the `hash` function is replaced by an N-deep recursion over `hash`, like:

$$\begin{aligned} &\texttt{rhash}(p\|s, 8V, N) \\ &= \texttt{hash}(\texttt{hash}(\ldots\texttt{hash}(p\|s, 8V), \ldots, 8V), 8V) \end{aligned}$$

then, if `hash` is not broken, (1) becomes:

$$2^{H(p)}\Big(N\,\texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k)\Big) \tag{3}$$

and (2) becomes:

$$\begin{aligned} 2^{H(p)}(N+0) &= N2^{H(p)} \\ &= 2^{H(p)+\log_2 N} \end{aligned} \tag{4}$$

at this point, thanks to cryptographic guarantees concerning properties of hashing functions, there is absolutely no security distinction between a password with shannon's $H(p)+\log_2 N$ entropy bits, and a password with just $H(p)$ entropy bits that made use of the N-deep recursive calls of `hash`.

shannon's entropy of p remains $H(p)$, but thanks to the recursive calls of `hash`, that password will be as expensive as another password $\hat{p}$, such that $H(\hat{p}) = H(p) + \log_2 N$.

i think it will be simpler if we introduce the function-dependent caveman's entropy C as a measure. it goes like this:

$$C\Big(p, \texttt{hash}(\ldots)\Big) = H(p) \tag{5}$$

$$C\Big(\hat{p}, \texttt{hash}(\ldots)\Big) = H(p) + \log_2 N \tag{6}$$

$$\begin{aligned} C\Big(p, \texttt{rhash}(\ldots, N)\Big) &= H(p) + \log_2 N \\ &= H(\hat{p}) \end{aligned} \tag{7}$$

security-wise, there is no distinction between the more complex password $\hat{p}$, and the simpler password p that used $\texttt{rhash}(\ldots, N)$. so i really think we need to measure password security in C instead of H.

2.2 memory-hard `hash`

let `mhash` be like `rhash`, except that it also requires M many memory bytes such that, as available memory is linearly reduced from M, penalty in cpu time grows exponentially. let M be requested memory, $\hat{M}$ be available memory, and $e(M - \hat{M})$ be the exponential penalty value for reduction in memory, where $e(0) = 1$.

$$\begin{aligned} &\texttt{cost}\Big(\texttt{mhash}(p\|s, N, M)\Big) \\ &= \texttt{cost}\Big(\texttt{rhash}(p\|s, N)\Big)^{e(\hat{M}-M)} \end{aligned} \tag{8}$$

if `hash` in (1) is replaced by the M-bytes memory-hardened N-deep recursion hash function `mhash`, then (1) becomes:

$$2^{H(p)}\Big(N^{e(M-\hat{M})}\,\texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k)\Big) \tag{9}$$

(2) becomes:

$$\begin{aligned} 2^{H(p)}(N^{e(M-\hat{M})} + 0) &= N^{e(M-\hat{M})}2^{H(p)} \\ &= 2^{H(p)+\log_2 N^{e(M-\hat{M})}} \\ &= 2^{H(p)+e(M-\hat{M})\log_2 N} \end{aligned} \tag{10}$$

and caveman's entropy becomes:

$$C\Big(p, \texttt{mhash}(\ldots, N, M)\Big) = H(p) + e(M-\hat{M})\log_2 N \tag{11}$$

2.3 the perfect lie theorem

let p be a password with $H(p)$ shannon's entropy bits. let $\hat{p}$ be a more complex password with $H(p)+e(M-\hat{M})\log_2 N$ shannon's entropy bits, where M, $\hat{M}$ and N are all positive numbers.

then caveman's entropy says that the following keys are information theoretically indistinguishable for as long as only p and $\hat{p}$ remain unknown (everything else is known, such as the distribution from which p and $\hat{p}$ was sampled), and for as long as `hash` is not broken:

- $k \leftarrow \texttt{mhash}(p\|s, N, M)$
- $\hat{k} \leftarrow \texttt{hash}(\hat{p}\|s)$

in other words:

$$C\Big(p, \texttt{mhash}(\ldots, N, M)\Big) = H(\hat{p}) \tag{12}$$

since the assumption that passwords are kept away from the adversary is fundamental in a symmetric encryption context, i think it makes since that we measure our security with memory-hard key derivation functions using the caveman's entropy C instead of shannon's entropy H.

from a security point of view, it will feel absolutely identical to as if the password got injected with extra shannon's entropy bits. no one can tell the difference for as long as the fundamental assumption of hiding passwords is honoured, as well as the hashing function `hash` is not broken.

in other words, we can say, if password p is unknown, and `hash` is not broken, then we have injected into p extra shannon's entropy bits. this lie will be only discovered after p is revealed.

if you think that it is impossible for this *lie* to be *truth* under the secrecy of p, then i've done an even better job: proving that cryptographically secure hashing functions do not exist. likewise, same can be trivially extended to: cryptographically symmetric ciphers do not exist.

so you have to pick only one of these options:

1. either accept that the lie is truth. i.e. accept that we've injected shannon's entropy bits into p, for as long as only p is not revealed.
2. or, accept that cryptographically-secure hashing and symmetric-encryption functions functions do not exist.

theorem 1 (the perfect lie)**.** *when p is secret and* `hash` *is not broken, then shannon's entropy H of the derived key equals caveman's entropy C.*

i call theorem 1 the *perfect lie theorem* in a sense that a perfect lie is indistinguishable from truth.

the reason this lie is appealing is because it simplifies our quantification of the amount of security that we have gained by using a given key derivation function, such as `rhash` or `mhash`.

without treating this lie as truth, our only hope would be surveying the asics industry. but with this lie, we have one more approach to get a feel of the gained security quantity by just accepting caveman's entropy C as shannon's entropy H, and move on as if the lie is truth, and no one can notice it.

we can also look at it from the perspective of *occam's razor*. i.e. if two things are not distinguishable from one another, then assuming that they are just the same thing is simpler than assuming otherwise.

to be more specific about *occam's razor*: (1) each assumption bit has a positive probability of error by definition, (2) since assuming that indistinguishable things are different than one another is more complex (i.e. more assumption bits) than assuming not, and (3) since there is no observable difference between the two things, therefore it necessarily follows that our model's total error will be reduced if we accept that the indistinguishable things are identical (i.e. which is what theorem 1 says).

3 ciphart

3.1 parameters

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 *caveman's entropy bits* to inject into p.
K	output key's size in bytes.

3.2 internal variables

`enc`	encryption function.
`hash`	hashing function.
C	$\leftarrow \begin{cases} 64 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes} \\ \ldots \end{cases}$ this to reflect the block size of the encryption algorithm that implements `enc`.
V	$\leftarrow \begin{cases} 32 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-128} \\ 32 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-256} \\ \ldots \end{cases}$ this is the size of the encryption key that's used to solve *ciphart*'s tasks. this is different than the `enc`-independent K 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 *caveman's entropy bits* that were injected into p, 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 summary