ciphart

memory-*harder* key derivation with easier security interpretation

caveman[1]

2021-02-17 06:13:10+00:00

synopsis— argon2[2] is mostly nice, as it is a simple memory-hard key derivation function relative to *scrypt*, which makes understanding what it is doing easier. but i argue that *argon2* is still not nice enough, as it can be simpler and harder.

currently, if you want to know what *argon2* is giving for your security-wise, you need to survey the industry of application-specific integrated-circuits (asics) in order to obtain a cost-money map, as done in the *scrypt* paper. the housekeeping of this approach is too expensive as the industry is constantly changing, plus it remains a rough estimate with inadequate theoretical guarantees.

henceforth, i propose *ciphart* — a memory-*harder* key derivation function, with a security contribution that is measured in the unit of shannon's entropy. i.e. *ciphart* can say that it is injecting x many shannon's entropy bits to your password.

- the simpler security interpretation is thanks to *the perfect lie* theorem that i discovered, which allows *ciphart* to claim that it is injecting shannon's entropy bits into input passwords, for as long as the password remains a secret.

  the password remaining a secret for the adversary is a fundamental assumption of password-based security, hence the theorem applies against the vestiary, which is the side that it matters.

- the memory-harderness allows to require ridiculously large amounts of memory, way beyond our random-access memory. this is done by utilising hard-disks, and can be done conveniently, in part, thanks to my discovery of the fact that caching derived keys into disks, is practical for almost every use cases.

  it's optional, but i personally use it as it significantly enhances my security, and an adversary won't compromise my security, even if he steals my hard-disk.

`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] 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`

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

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

[6] see the *scrypt* paper for an example.

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.

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 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.

i like this shannon's entropy-based simplified security quantity, so i'm going to build on it.

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:

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

(2) becomes:

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

and the $\log_2$-scaled becomes $H(p) + \log_2 N$.

at this point, thanks to cryptographic guarantees concerning properties of hashing functions, there is absolutely no security distinction from adversary's point of view 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:

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

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

[7] statistically by confidence earned through peer review and attempts to break encryption algorithms.

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

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:

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

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[8])**.** *when p is secret and* `hash` *is not broken, then shannon's entropy H of the derived key equals caveman's entropy C.*

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).

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

4 caching derived keys

discovery 1. *caching keys securely is easily doable, and great security utility exists in doing so for expensively-derived keys.*

4.1 why it works

- when expensively derived keys are cached, only the first key derivation call will be expensive, while subsequent calls will be semi-instantaneous. this effectively allows users to tolerate much more expensive, or secure, key derivation as it only happens during the initial, say, login phase. subsequent use of the extremely expensive key is instantaneous.

  so instead of having the user use a somewhat expensive key derivation by wayting say, 3 seconds in each login, he will —instead— wait, say 10 seconds in his initial login in order to utilise a much more expensive key derivation, and then wait near 0 seconds for every subsequent login as the expensive key is cached.

- derived keys can be cached securely, without increasing most users' assumptions. e.g. cached keys can live in a *dm-crypt* partition that is encrypted with a large encryption key that is stored properly, and the cache can have strict read permissions so that only the unique user that runs `ciphart` executable can read it.

  this only requires to trust the user `root`, which is already trusted by almost everyone. so we are not introducing a new assumption.

for most people, if `root` is compromised, then the adversary can break every other key derivation function, including those that do not cache keys, by simply, say, running a keylogger.

so, practically, we are not increasing the assumptions, but we are only increasing the value that we can extract from the assumptions that we are already following.

most importantly, utilising discovery 1 allows us to achieve a memory-*harder* key derivation in an extremely usable way. more details on memory-harderness later.

4.2 potential adversary strategies

let's see what can the adversary may try to do against a key caching system:

- **adversary strategy 1:** hack into user's system and execute a program as that user that tries to read the password cache file to obtain the memory-harder key inside it.

  answer: he will not be able to read the file due to strict read permissions of the cached files as set earlier.

- **adversary strategy 2:** steal user's hard disk and try to mount it in his system, to login as root and chance permissions of the key cache files.

  answer: the partition where the cached files are saved are properly encrypted, so he can't see the cached files, let alone changing their read permissions.

- **adversary strategy 3:** break into machine's `root` account.

  answer: he will succeed, but then he can also run a keylogger, which will also break every other key derivation function, even those who do not use key caching, such as *scrypt*, *argon2*, etc.

4.3 user scenario example

user has a password manager to generate unique 256 bit entropy keys for each service that he uses. but user doesn't want to decrypt the password manager's database using a physical dongle, so he cannot use a high-entropy key for the purpose of unlocking the password manager.

why? several possible reasons. maybe (1) he tends to lose or forget dongles and he doesn't want the delay associated with resolving such physical dongle issues, maybe (2) not having a physical dongle will enhance his repudiation case, or maybe (3) he wants to be torture-resistant so he doesn't want to let the adversary take the physical dongle from him by force[9].

in this case, the user memorises a sensible password that he can remember, with enough initial entropy, and then uses *ciphart* with disk caching to inject large amounts of entropy bits into his derived keys, way beyond the reach of pre-*ciphart* key derivation functions; thanks to theorem 1 and discovery 1.

5 ciphart

5.1 parameters

p	password.
s	salt.
M	total random-access memory in bytes.
D	total hard-disk memory in bytes.
L	number of memory lanes for concurrency.
T	number of tasks per lane segment.
S	number of lane segments per hard-disk read.
B	minimum *caveman's entropy bits* to inject into p.
K	output key's size in bytes.

[9] when elon musk's neural-link matures enough, we may end up getting keys, and other information, pulled from our heads by force. but, until then, our brain seems to be a pretty secure information store.

5.2 internal variables

`enc`	encryption function.
`hash`	hashing function.
C	$\leftarrow \begin{cases} 64 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes} \\ \dots \end{cases}$ 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[10].
$\hat{T}$	$\leftarrow \max(\lceil VC^{-1} \rceil, T)$. 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.
$\hat{D}$	$\leftarrow D - (D \bmod C) + C$. this is to ensure that it is in multiples of block 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	$\leftarrow *\texttt{hash}(p \,\|\, s, V)$. a pointer to the first byte where V-bytes key is stored.

5.3 output

k	K-bytes key.
$\hat{B}$	actual *caveman's entropy bits* that were injected into p, where $\hat{B} \geq B$.

5.4 steps

steps of *ciphart* is shown in algorithm 1. this corresponds to *argon2d*. adding a *ciphart-i* variant is a trivial matter,

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

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, ..., G − 1 do
3          for l = 0, 1, ..., L − 1 do
4              for t = 0, 1, ..., T − 1 do
5                  i ← gLT + lT + t;
6                  if t < T − 1 then
7                      j ← i + 1;
8                  else if t = T − 1 then
9                      j ← i − T + 1;
10                 m_j ← enc(m_i, n_l, v);
11                 n_l ← n_l + 1;
12                 if f = 0 then
13                     v ← m_j mod (gLTC + tC − V);
14                     if v ≥ gLTC − V then
15                         v ← v + lTC;
16                 else
17                     v ← m_j mod (M̂ − LTC + tC − V);
18                     if v ≥ gLTC + tC − V then
19                         v ← v + LTC;
20                     else if v ≥ gLTC − V then
21                         v ← v + lTC;
22         N ← N + LT;
23         if N ≥ 2^B then
24             g_last ← g;
25             go to line 27;
26     f ← 1;
27 i ← g_last LT;
28 k ← hash(m_{i+0T} ‖ m_{i+1T} ‖ ... ‖ m_{i+(L−1)T}, K);
29 B̂ ← log_2 N;
30 return k, B̂
```

6 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.

7 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*. □

8 memory-*harder*ness

thanks to discovery 1, memory-harderness is possible. this process goes like this:

1. starts by recursively encrypting D many bytes of some predefined sequence, such as zeros or `0xdbdb`..., using a key derived from an cheaply-hashed password p. by the end of this step, we have D many key-based random sequence saved in the an encrypted hard disk partition.

2. update the derived key to be the last V bytes in the D bytes file.

3. move on to run a variant of *argon2*, except that, every time S many random-access memory lane segments are completed, some V many bytes from D are chosen randomly, not only based on the content the random-access memory, but also based on the D bytes. this step makes *ciphart* require $D + M$ bytes.

4. delete the D bytes, just to free up the disk space.

if S is large enough, this can be done efficiently without blocking the cpu noticeably, as the randomly obtained V bytes can be read by using the $O(1)$ operation `seek` over the D bytes.

9 comparison