ciphart

faster memory-*harder* key derivation with easier security interpretation

caveman[1]

2021-03-02 20:43:19+00:00

synopsis—*argon2*[2] is a fast and simple memory-hard key derivation function. compared to *scrypt*[3], *argon2* is better, specially for its simplicity. but i argue that *argon2* is not fast enough, not memory-hard enough, and its contribution to our security is not simple enough to understand.

so i propose *ciphart*, which is:

- easier — because its security contribution is measured in the unit of shannon's entropy. i.e. when *ciphart* derives a key for you, it tells you that it has *injected* a specific guaranteed quantity of shannon's entropy bits into your derived key. this is possible thanks to my invention, the "perfect lie" theorem.

  this offers a great help as it gives us yet another much simpler approach to quantify our security gain as opposed to being limited to surveying the industry of application-specific integrated circuits as done in the *scrypt* paper.

- harder — because it can require crazy-large amounts of memory, beyond our random-access memory, thanks to it being able to use the hard-disk as well. this is possible thanks to my discovery "cacheable keys".

  this is optional, but i extremely like it as it effectively gives me much more security while eventually becoming much faster as well, and the adversary cannot get my cache even if he steals my hard-disks.

- faster — because it does not abuse hashing functions. it uses hashing functions when using them is more suited, and uses encryption functions when encryption functions are more suited. this is thanks to my "simplification" attempt.

  argon2 incorrectly limits itself to use only a hashing function. at the surface it may appear simpler, but it is actually more wasteful of resources, requires reinventing more wheels, and needlessly increases function's total assumptions.

`libciphart`[4] 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`[5] 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] `http://www.tarsnap.com/scrypt/scrypt.pdf`

[4] `https://github.com/Al-Caveman/libciphart`

[5] `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[6] 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.*

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

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 bruteforced within a target time frame[7]. 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.

another way is to ignore anything that has no cryptographic guarantee. so, in (1), cryptography guarantees[8] 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 fundamental ideas

2.1 caveman's entropy

2.1.1 recursive `hash`: `rhash`

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:

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

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

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

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

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.

2.1.2 memory-hard `hash`: `mhash`

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, A be available memory, and $e(M-A)$ be the exponential penalty value for reduction in memory, where $e(0) = 1$.

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

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

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

(2) becomes:

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

and caveman's entropy becomes:

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

2.2 "perfect lie" theorem

let p be a password with $H(p)$ shannon's entropy bits. let $\hat{p}$ be a more complex password with $H(p) + e(M-A)\log_2 N$ shannon's entropy bits, where M, A 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 (perfect lie[9]). *for as long as password p remains a secret and* `hash` *is not broken, derived keys $k_1 = \texttt{rhash}(p, \ldots, N)$ and $k_2 = \texttt{mhash}(p, \ldots, N, M)$ will have their shannon's entropies, $H(k_1)$ and $H(k_2)$, increase beyond $H(p)$ in order to equate their caveman's entropies $C(p, \texttt{rhash}(\ldots, N))$ and $C(p, \texttt{mhash}(\ldots, N, M))$, respectively.*

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).

2.3 "cacheable keys" discovery

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

2.3.1 why does it work

- 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 already have.

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.

2.3.2 potential adversary strategies

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

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

- **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.

2.4 "simplification" attempt

which one is simpler when, say, building a wooden house?

- option 1 — use only a nail and, when you need a screw, modify the nail into a screw.
- option 2 — use a nail and a screw.

on the surface, option 1 may appear as the simpler choice as it only uses nails, while *option 2* uses both nails ans crews.

but a deeper look shows that option 1 is actually a lie, as it is also using a nail and a screw, except that the screw is re-invented by modifying a nail. so, my answer is that *option 2* is the simpler choice.

similarly, i think *argon2*'s use of only a hashing function is rather a lie, as it effectively ends up re-inventing a symmetric block encryption algorithm off a hashing function for as far as a key derivation function is concerned. here is how:

- when *argon2* uses a hashing function to compress potentially large amounts of data to derive the final output key, that's a fine use of a hashing function, because one of the tasks of a hashing is to do such compression.
- but when *argon2* uses a hashing function to solve tasks in the memory pad, it tries to use a hashing function to emulate the behavior of a symmetric block encryption function, for as long as a key derivation function is concerned.

  i.e. when *argon2* concatenates previous task's hash against a randomly picked task's hash in order to compute next task's hash, it is effectively trying to re-invent the idea of a pre-shared key, for as long as a key derivation function is concerned.

  why so? because there is no compression here, so why use a hashing function? why not use a symmetric block cipher, and randomly pick the key instead of emulating a key using concatenation?

but why bother? because:

- re-inventing things are more expensive than not.
- hashing functions are generally more expensive than symmetric block encryption functions. i think this is because hashing functions also have the duty of compression, shile symmetric block encryption functions don't.

attempt 1 (simplification)**.** *use hashing functions only when compression happens. when no compression takes place, use a block encryption function instead.*

3 ciphart

3.1 the algorithm

3.1.1 parameters

P	password.
S	salt.
M	total random-access memory in bytes.
D	total hard-disk memory in bytes.
F	temporary file's path.
Y	whether key caching is enabled.
L	number of memory lanes for concurrency.
T	number of tasks per lane segment.
H	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.

3.1.2 internal variables

`enc`	encryption function.
`hash`	hashing function.
`read`	hard-disk file reading function, with seeking. e.g. `read`(x, y, z) reads z many bytes from file x after seeking y bytes forward.
`write`	hard-disk file writing 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 output key's size, K, which is `enc`-independent.

$\hat{T}$ $\leftarrow \max(\lceil VC^{-1} \rceil, T)$. 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 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.

n $\leftarrow 0$. actual number of times `enc` is called.

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 counter indicating number of times memory is filled with $\hat{M}$ many bytes.

d $\leftarrow 0$. a counter indicating total number of saved blocks into hard-disk.

h $\leftarrow 0$. a counter indicating number of processed lane segments since the last hard-disk read.

q $\leftarrow 0$. a counter indicating number of times key was updated from the hard-disk.

$*v$ $\leftarrow *\texttt{hash}(P\|S\|M\|D\|\ldots\|K, V)$. a pointer to the first byte where V-bytes key is stored. v is the key itself, and $*v$ is a pointer to it.

Z $\leftarrow \texttt{hash}(v\|0, V)$. file name where output key, k, is expected to be cached, if k was previously cached.

3.1.3 output

k K-bytes key.

n total number of times `enc` was actually called. $\log_2 n$ is total number shannon's entropy bits that *ciphart* injected into k, such that $\log_2 n \geq B$.

3.1.4 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  if Y and exists(Z) then
2      k, n ← read(Z, 0, V + sizeof(n));
3      return k, n
4  while 1 do
5      for g ← 0, 1, ..., G − 1 do
6          for l ← 0, 1, ..., L − 1 do
7              for t ← 0, 1, ..., T − 1 do
8                  i ← gLT + lT + t;
9                  if t < T − 1 then
10                     j ← i + 1;
11                 else if t = T − 1 then
12                     j ← i − T + 1;
13                 m_j ← enc(m_i, n_l, *v);
14                 n_l ← n_l + 1;
15                 if f = 0 then
16                     *v ← m_j mod (gLTC + tC − V);
17                     if *v ≥ gLTC − V then
18                         *v ← *v + lTC;
19                 else
20                     *v ← m_j mod (M̂ − LTC + tC − V);
21                     if *v ≥ gLTC + tC − V then
22                         *v ← *v + LTC;
23                     else if *v ≥ gLTC − V then
24                         *v ← *v + lTC;
25         n ← n + LT;
26         if d ≤ D then
27             for i ← gLT, ..., gLT + (T − 1) do
28                 write(F, m_i);
29                 d ← d + 1;
30                 if d ≥ D then
31                     break;
32         else
33             h ← h + L;
34             if h ≥ H then
35                 *v ← *read(F, v mod (d − V), V);
36                 q ← q + 1;
37                 h ← 0;
38             if f ≥ 2 and q ≥ 1 and n ≥ 2^B then
39                 g_last ← g;
40                 go to line 42;
41     f ← f + 1;
42 i ← g_last LT;
43 k ← hash(m_{i+0T} ‖ m_{i+1T} ‖ ... ‖ m_{i+(L−1)T}, K);
44 if Y then
45     write(Z, v‖n);
46 delete(F);
47 return k, n
```

3.2 noteworthy features

3.2.1 parallelism

since iterations of the loop in line 6 in algorithm 1 are fully independent of one other, they can quite happily utilise L cpu cores, specially when segment sizes, T, are larger.

3.2.2 memory-hardness

Proof. algorithm 1 is just a variation of *argon2d*, except that it uses an encryption function, `enc`, instead of a hashing function. so if *argon2d* is memory-hard, then so is *ciphart*. □

3.2.3 memory-*harder*ness

thanks to discovery 1, we can cache keys, as done in line 1, without increasing assumptions of the threat model of the vast majority of users. then, memory-*harder*ness becomes possible. this process goes like this:

1. starts by running an `enc`-based variant of *argon2*, except that, as it is going, it keeps writing the updated segments into the hard-disk until the size of it satisfies the D bytes limit. this is shown in line 26 in algorithm 1.

   optimising this hard-disk filling with D bytes is not a big deal, since this feature is probably going to be used only once; thanks to key caching. that said:

   - this feature is optional. i.e. in case someone doesn't like the hard-disk caching, he can set $D \leftarrow 0$ to disable it. but, for most people, i don't understand why you would want to disable it. e.g. if you're already trusting `root`, then i think that you can use this feature without changing your threat model.

     i personally like it a lot as it allows me to achieve memory-*harder*ness way beyond my random-access memory. just imagine the look on the face of those asics crackers once they hear that your *ciphart* requires, say, 50 giga bytes!

   - when key caching is enabled, i.e. $Y \leftarrow 1$, this hard-disk writing is done only initially, and subsequent uses appear as almost instantaneous.

   - this hard-disk writing can be slightly optimised by using a non-blocking write operation. but i think that the trivial reduction in this time may not justify increasing code's complexity, so i don't plan to implement non-blocking writes in `libciphart`.

2. then, once the D bytes hard-disk requirement is satisfied, the process continues to run the modified `enc`-based variant of *argon2* except that it updates the key v from those D bytes every time S many segments are solved. this step makes *ciphart* require $D + M$ bytes. this is shown in line 34 in algorithm 1.

   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.

3. delete the D bytes, just to free up the disk space. no need to securely delete those D bytes since we can save them using a temporary random key that's forgotten later on.

3.3 comparison

4 application scenarios

4.1 a currently-useful scenario

user has a password manager which generates unique 256 bit entropy keys for each online service that he uses. the user also renews keys for his online services every now and then. so his online accounts generally have high security.

but user's problem is how to lock and unlock his password manager's passwords database. should he use a physical usb-stick key that types a high-entropy key that encrypts and decrypts the database? the user doesn't want this physical key because of several reasons:

- he tends to lose his keys a lot, and, for certain tasks, the risk of needing to wait for until he gets a backup usb-stick key is too much.

- he doesn't want to be caught having cryptographic usb-stick keys, because as such is an evidence that he has encrypted content. the user wants to have the choice to lie that he has no clue. so not having usb-stick keys with him helps his case to lie.

- he wants to be torture-resistant so that an adversary cannot forcefully take his keys from him in order to login into his services. he may rather want to die than to give the password to the adversary. this works because, so far, the brain is a pretty private information store.

in this scenario, the user memorises a sensible password that he can remember, with enough initial entropy, and then uses *ciphart*, preferably 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.

here, the expensively derived key is only used to unlock a local password manager, which offers a protection against situations where a backup copy of the passwords database is stolen. this may enable the user to store his passwords manager in an online file synchronisation service for more convenient system migrations to further reduce his login delays should he face the need to migrate to a new, say, personal computer.

advertisement 1. *in case you're interested in such a passwords manager, i've also made* nsapass[10] *— a flexible and a simple passwords manager in a few hundreds of python lines of code, that uses* ciphart *by default. i think this is the best command-line interfacing passwords manager by far, for its usability, and for the fact that auditing it is easy, thanks for it being only in a few hundreds of python lines.*

[10] `https://github.com/Al-Caveman/nsapass`

4.2 a later-useful scenario

all input password fields will internally call `libciphart` to derive more expensive keys. this way, applications, such as *firefox*, *mutt*, ..., will never send actual passwords, but will only send *ciphart*-derived keys with increased shannon's entropy content.

thanks to theorem 1, this will automatically increase shannon's entropy of all users' passwords without requiring users to memorise harder passwords. thanks to discovery 1, the user will not face any delay in his daily use, except only an initial delay to create the cache entry.

i also think that it is *generally* better to have expensive key derivation functions in the client side as opposed to the server side. because remote servers always have the incentive to reduce the complexity of the key derivation function in order to free more resources for other things that bring them money.

either way, *ciphart* is perfectly usable on the server side. it is just that i think *ciphart*, *argon2*, *scrypt*, ... make better sense when placed on the client side.

A donations

this work is sponsored by a bright nerdiness that has unexpectedly sparked somewhere in an endless dark space. nothing here was supposed to happen. the cause remains unknown, so we call it *something random with large entropy*.

ancient mythology has it that, every time a donation is made for a good cause, a beautiful torch is kindled somewhere deep within an otherwise cold and dark space.

the torch is pure with an innocent smile every time a spending is committed towards good. but, she drops a tear when the spending is done in excess. despite her attempts to maintain a steady light, sometimes her tears force her light to flicker, and sometimes the flickering causes her to disappear. when she is gone, she is never back again, except in the memories of those who have once witnessed her charm...

so, if you opt for a donation, i'd appreciate exercising wisdom, so that the sensible balance is not exceeded; lest we want a beautiful torch to be born, only to make her disappear in her tears.

A.1 bitcoin

`bc1qtylzjtgd0yu4v7f8hyfzufn7nu692v9fc88jln`