# ciphart

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

caveman[1]

2021-03-17 05:01:15+00:00

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

henceforth, 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 symmetric block encryption functions when using them is more suited. this is thanks to my "hashing is only for compression" law.

  *argon2* incorrectly limits itself to only use a hashing function. at the surface it may appear simpler, but it is actually more complex as it ends up re-inventing what resembles a symmetric block encryption function off the hashing function, except for being slower and with needless potential entropy loss.

`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

## 1.1 passwords' entropy

**definition 1.1** (options set)**.** *$\mathcal{O}_x$ is the set of options that $x$ might be one of.*

say that $\mathcal{O}_x$ is the options set of thing $x$. generally, a scalable way to find which of those options is $x$, is to ask questions, such that each time one of them is answered, the quantity of considered options shrinks in half or more. let's stick to shrinkage in half; i.e. balanced binary trees. this means that the total number of such balanced binary questions is $\log_2 |\mathcal{O}_x|$, where $|\mathcal{O}_x|$ is the quantity of total options in $\mathcal{O}_x$.

but is that scalable approach, also the most efficient one in finding $x$? the answer is: it depends. the reason is that a binary search tree is not considering the probability of a given option being $x$. e.g. which one is better:

- ask a question that would reveal that, say, 10 options were not $x$, in a way that we were not surprised as, say, $\Pr(o = x) \leq 0.0001$ for any $o \in \{o_0, o_1, \dots, o_{10-1}\}$? we already knew this, so getting these 10 options revealed wasn't really informative for us, hence asking this question was sort of a waste of time.

- ask a question that would reveal that, say, 5 options in $\mathcal{O}_x$ were not $x$, in a way that would completely blow our minds as we thought that one of them would be $x$ as, say, $\Pr(o = x) \geq 0.9999$ for any $o \in \{o_0, o_1, \dots, o_{5-1}\}$? we didn't expect this, so knowing that these highly likely options were not $x$ was quite informative.

if we take such probabilities into account, we end up using a possibly imbalanced binary tree in such a way that would maximise our information gain every time a question gets answered. because it takes such probabilities into account, we usually end up knowing more about what $x$ is, or what it is not, every time a question gets answered, than a balanced binary tree that doesn't take such probabilities into account. effectively, we will end up knowing what $x$ is with less questions, asymptotically on average.

shannon's entropy tells us the minimum number of questions to be asked, asymptotically in average, in order to extract all information about $x$, while also considering the probability of each option being $x$:

$$H(\mathcal{O}_x) = \sum_{o \in \mathcal{O}_x} \Pr(o = x) \log_2 \Pr(o = x)^{-1} \tag{1}$$

so when is $\log_2 |\mathcal{O}_x|$ optimal? $H(\mathcal{O}_x) = \log_2 |\mathcal{O}_x|$ if revealing any option is equally informative to us compared to revealing every other option, which is the case when there is no redundancy in the options set. i.e. when, for any $o \in \mathcal{O}_x$, $\Pr(o = 0) = |\mathcal{O}_x|^{-1}$.

*Proof.*

$$\begin{aligned} H(\mathcal{O}_x) &= \sum_{o \in \mathcal{O}_x} |\mathcal{O}_x|^{-1} \log_2 (|\mathcal{O}_x|^{-1})^{-1} \\ &= \sum_{o \in \mathcal{O}_x} |\mathcal{O}_x|^{-1} \log_2 |\mathcal{O}_x| \\ &= |\mathcal{O}_x| |\mathcal{O}_x|^{-1} \log_2 |\mathcal{O}_x| \\ &= \log_2 |\mathcal{O}_x| \end{aligned} \tag{2}$$

□

let's say that $p$ is an unknown password that the adversary got its $8V$ bits key $k \leftarrow \texttt{hash}(p, 8V)$, and that he wants to find $p$ that gave $k$. also say that the adversary knows $\mathcal{O}_p$ and the distribution by which $p$ was sampled according to. what this means is that:

- if the distribution is uniform random, then, asymptotically on average, the adversary would need to ask $\log_2 |\mathcal{O}_p|$ many balanced binary questions until he finds out $x$. i.e. $H(\mathcal{O}_p) = \log_2 |\mathcal{O}_p|$.

- if the distribution is not uniform random, then, asymptotically on average, the adversary would need to ask less questions than $\log_2 |\mathcal{O}_p|$, as the unlikely options would be usually not asked about because of the fact that the binary questions are imbalanced to be optimised for the non-uniform random probability distribution. i.e. $H(\mathcal{O}_p) < \log_2 |\mathcal{O}_p|$.

if the adversary knows $p$'s entropy $H(\mathcal{O}_p)$, it means that, asymptotically on average, he will need to ask $H(\mathcal{O}_p)$ binary questions in order to find $p$.

but these binary questions are theoretical, and may not exist in reality. specially for password hashes, it wouldn't make sense to ask "is $p \geq$ password123?", because passwords are non-numerical but categorical values, and because password hashes change completely for any change in the password, so ranges do not apply. so we cannot ask a single binary question to test a range of hashes. instead, we are forced to test every candidate password $\hat{p}$ by hashing it individually into $\hat{k} \leftarrow \texttt{hash}(\hat{p}, 8V)$, and testing whether $\hat{k} = k$.

so, if $H(\mathcal{O}_p)$ is only a theoretical number of questions, which we cannot ask in the case of passwords hashing, then why do we use this number? the answer is, to obtain the total number of individual candidate tests that we need to perform asymptotically on average, because this number is $2^{H(\mathcal{O}_p)}$. i.e. the adversary would end up, asymptotically on average, testing no less than $2^{H(\mathcal{O}_p)}$ many password candidates.

**lemma 1.1** (walking backwards to entropy). *in the context of password brute-forcing, if, asymptotically on average, the minimum number of options that need to be tested to find a target key is $x$ many options, then we know that the entropy of that target key is $\log_2 x$.*

## 1.2 passwords' security

**definition 1.2** (systems' security). *the security of a system is the cost of the cheapest method that can break it.*

### 1.2.1 `hash`

say that we've got an $8V$ bit key $k \leftarrow \texttt{hash}(p, 8V)$, derived from an unknown password $p$. say that the adversary has $k$ but wants to figure out $p$.

asymptotically on average, the adversary would need to hash at least $2^{H(\mathcal{O}_p)}$ many password candidates, and test each one of them against $k$. each test's cost is the cost of hashing a candidate password $\hat{p}$ into a candidate key $\hat{k}$, and the cost of testing whether $\hat{k} = k$. his total cost is:

$$2^{H(\mathcal{O}_p)} \left( \texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k) \right) \tag{3}$$

one way to estimate the `cost` function is to survey the asics industry to get an idea how much money it costs to get a given key space, or password space, brute-forced within a target time frame[6]. the housekeeping of this approach is expensive, 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 (3), cryptography guarantees[7] that $2^{H(\mathcal{O}_p)}$ many `hash` calls must be 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 it's so cheap it's easier to just assume that its cost is free. this way (3) becomes just:

$$2^{H(\mathcal{O}_p)}(1+0) = 2^{H(\mathcal{O}_p)} \tag{4}$$

further, for convenience i guess, it seems that people report it in the $\log_2$ scale. i.e.:

$$\log_2 2^{H(\mathcal{O}_p)} = H(\mathcal{O}_p) \tag{5}$$

i think this is why people use shannon's entropy of passwords as a measure of their security. not because it is the quantity of security, but rather because its the quantity of *simplified* security.

i like using shannon's entropy as a measure of simplified security quantity, so i'm going to build on it.

### 1.2.2 recursive `hash`: `rhash`

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

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

then, if `hash` is not broken, (3) becomes:

$$2^{H(\mathcal{O}_p)} N \left(\texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k)\right) \tag{6}$$

(4) becomes:

$$\begin{aligned}2^{H(\mathcal{O}_p)} N(1+0) &= N 2^{H(\mathcal{O}_p)}\\ &= 2^{H(\mathcal{O}_p)+\log_2 N}\end{aligned} \tag{7}$$

and the $\log_2$ scaled version becomes:

$$H(\mathcal{O}_p) + \log_2 N \tag{8}$$

### 1.2.3 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 $e(M-A) = 1$ if $M - A \leq 0$.

$$\begin{aligned}&\texttt{cost}\left(\texttt{mhash}(p, N, M)\right)\\ &= \texttt{cost}\left(\texttt{rhash}(p, N)\right)^{e(M-A)}\end{aligned} \tag{9}$$

if `hash` in (3) is replaced by the $M$-bytes memory-hardened $N$-deep recursion hash function `mhash`, then (3) becomes:

$$2^{H(\mathcal{O}_p)} N^{e(M-A)} \left(\texttt{cost}(\texttt{hash}) + \texttt{cost}(\text{if } \hat{k} = k)\right) \tag{10}$$

(4) becomes:

$$\begin{aligned}2^{H(\mathcal{O}_p)} N^{e(M-A)}(1+0) &= N^{e(M-A)} 2^{H(\mathcal{O}_p)}\\ &= 2^{H(\mathcal{O}_p)+\log_2 N^{e(M-A)}}\\ &= 2^{H(\mathcal{O}_p)+e(M-A)\log_2 N}\end{aligned} \tag{11}$$

and $\log_2$ scaled version becomes:

$$H(\mathcal{O}_p) + e(M-A)\log_2 N \tag{12}$$

### 1.2.4 caveman's entropy

**definition 1.3** (caveman's entropy)**.** *it is clear that (5) is shannon's entropy, but i didn't make it clear yet what (8) and (12) are, so i will give them a temporary name until i tell you what they are later on in this paper. for now, let's call it* caveman's entropy*, $C$, which i define as follows:*

$$\begin{aligned}C\Big(p, \texttt{hash}(\dots)\Big) &= H(\mathcal{O}_p)\\ C\Big(p, \texttt{rhash}(\dots, N)\Big) &= H(\mathcal{O}_p) + \log_2 N\\ C\Big(p, \texttt{mhash}(\dots, N, M)\Big) &= H(\mathcal{O}_p) + e(M-A)\log_2 N\end{aligned}$$

# 2 fundamental ideas

## 2.1 "entropy injection" theorem

**definition 2.1** (option testing)**.** *an option is tested if a hash is calculated, and then checked for equality against a value.*

so, option testing is very physical. if the adversary ended up calculating more hashes of things, and then checked if the hashes matched some output, then he has effectively tested more options.

then, combining this fact with lemma 1.1, testing more options necessarily means that the adversary has suffered more entropy; aka needed to ask more questions.

the adversary cannot deny asking more questions by stating that his intention wasn't to ask them. his intentions are irrelevant. when he commits the physical action of hashing something and then testing it for equality, he has physically increased the number of questions that he has asked so far.

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

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

algorithm 1 shows an example of a function that uses cryptography to force the adversary to calculate more hashes and test them for equality. i.e. effectively forcing the adversary to ask more questions. in lines 3 and 8 the adversary is forced to calculate a hash, and in line 6 the adversary is forced to perform an equality test to see whether the calculated hash equals any of the elements in the set $\mathcal{H}$.

**algorithm 1:** $\texttt{irhash}(p, 8V, N)$

1 let $i = 0, j = 1$;
2 allocate $8V$ bits variables, $k_i$ and $k_j$;
3 $k_i \leftarrow \texttt{hash}(p, 8V)$;
4 let $\mathcal{H}$ be a set containing half the $8V$ hash space;
5 **for** $0, 1, \ldots, N-1$ **do**
6     **if** $k_i \in \mathcal{H}$ **then**
7         $k_i \leftarrow k_i + 1$;
8     $k_j \leftarrow \texttt{hash}(k_i, 8V)$;
9     $\hat{i} \leftarrow i$;
10     $i \leftarrow j$;
11     $j \leftarrow \hat{i}$;
12 **return** $k_j$

unless the function `hash` is broken, when the adversary wants to brute-force to find $p$ that gave $k$, he has no choice but to perform at least $N$ many hash calculations and at least $N$ many equality tests, which, by lemma 1.1 and definition 2.1, the adversary has effectively asked $\log_2 N$ many more theoretical binary questions in addition to $H(\mathcal{O}_p)$. so algorithm 1 effectively causes the entropy of the derived key $k$ to increase to:

$$H(\mathcal{O}_k) = H(\mathcal{O}_p) + \log_2 N \tag{13}$$

in other words, `irhash` in algorithm 1 is a variation of `rhash`, except that the former uses cryptography to inject $\log_2 N$ many entropy bits into the derived key.

`irhash` can be trivially extended to a memory-hard variant, `imhash`, which will give the following entropy:

$$H(\mathcal{O}_k) = H(\mathcal{O}_p) + e(M - A) \log_2 N \tag{14}$$

**theorem 2.1** (entropy injection)**.** *`irhash` and `imhash` inject $\log_2 N$ and $e(M-A)\log_2 N$ shannon's entropy bits into their derived keys, respectively, in addition to the entropy already in the input password $H(\mathcal{O}_p)$.*

if anyone rejects theorem 2.1, then he can consider this a proof by contradiction that that hashing functions, as well as any encryption function[8], do not exist. so, which one do you choose?

1. that i've proven that theorem 2.1 is injecting shannon's entropy bits?
2. or that i've proven by contradiction that hashing and encryption functions do not exist?

i personally think that i've proven (1), but if you think that i've proven (2), then that's nice too.

## 2.2 "perfect lie" theorem

equality testing is usually cheap. e.g. the `CMP` cpu assembly instruction usually takes 1 cpu cycle per cpu core, perhaps with a few extra cycles to copy data around.

on the other hand, each `hash` call may perform hundreds of cpu cycles. meaning the number of cpu cycles done by performing an equality test is relatively nothing compared to what `hash` is doing.

so, since the cycles due to the equality tests are so few, why not just ignore them, and lie that they are already done when calling `hash`?

**definition 2.2** (the lie)**.** *out of the total cpu cycles that are required to be performed by `hash`, 99% of them are to calculate the hash, and 1% of them are to test whether the calculated hash equals some desired hash. so, a single `hash` call is doing both: hashing and equality testing.*

as far as the security of a system in definition 1.2 is concerned, "the lie" is not distinguishable from truth, because either way what gives us the security is the required computations by cryptography, not necessarily what the computations mean. we can imagine that 99% of the computations done by `hash` mean calculating a hash, and imagine that 1% of them mean testing an equality, in order to allow ourselves to realise that what `rhash` and `mhash` are giving us are practically equivalent to shannon's entropy bits given by `irhash` and `imhash`.

so, if "the lie" is as good as not lying, except that not lying requires a more complex code base as shown in `irhash` in algorithm 1, or its memory-hard variant `imhash`, then why not lie and have a simpler code base? i'd say this lie is totally worth it.

**theorem 2.2** (perfect lie)**.** *for any password $p$, and any positive numbers $V$, $N$, $M$ and $A$, such that $M \geq A$:*

$$\begin{aligned} H(\mathcal{O}_{k_h}) &= C\Big(p, \texttt{hash}(\ldots)\Big) \\ &= H(\mathcal{O}_p) \\ H(\mathcal{O}_{k_r}) &= C\Big(p, \texttt{rhash}(\ldots, N)\Big) \\ &= H(\mathcal{O}_p) + \log_2 N \\ H(\mathcal{O}_{k_m}) &= C\Big(p, \texttt{mhash}(\ldots, N, M)\Big) \\ &= H(\mathcal{O}_p) + e(M-A)\log_2 N \end{aligned}$$

*where:*

$$\begin{aligned} k_h &= \texttt{hash}(p, 8V) \\ k_r &= \texttt{rhash}(p, 8V, N) \\ k_m &= \texttt{mhash}(p, 8V, N, M) \end{aligned}$$

[8] because encryption functions can be used to create hashing functions

thanks to the "perfect lie" theorem, we can now move on and use regular `rhash` and `mhash`, and —at the same time— dare to have a very simple security interpretation, of their contribution to our security, in the unit of shannon's entropy, without needing a more complex function such as `irhash` in algorithm 1.

## 2.3 "cacheable keys" discovery

**discovery 2.1** (cacheable keys)**.** *caching keys securely is easily doable, and great security utility exists in doing so for expensively-derived keys.*

**definition 2.3** (caveman's keys cache)**.** *a keys cache that has $O(1)$ asymptotic worst run-time complexity for insertion, deletion, and retrieval when the user knows the password. but, when the password is unknown, the cache has a brute forcing space of, at least, $n2^{H(\mathcal{O}_p)}$ many attempts, where $n$ is the size of the cache.*

*i.e. by lemma 1.1, the adversary has to suffer an entropy of $H(\mathcal{O}_p) + \log_2 n$, and $n$ can be made arbitrarily large by inserting fake cache entries, up to a point where the adversary would find no point in stealing user's hard disk. i.e. the adversary will find it as hard to brute force the key derivation function instead of caveman's keys cache.*

**theorem 2.3** (caveman's keys cache)**.** *caveman's keys cache exists.*

let's say that $k$ is an expensively derived key, say, by using *ciphart*, *argon2*, *scrypt*, etc. if we can securely cache $k$, it would mean that we can derive $k$ with more expensive parameters as it will be done only once, while subsequent calls of the key derivation function to obtain the key $k$ would be almost instantaneous as it will be pulled from the cache at $O(1)$ asymptotic worst run-time complexity.

it is trivial to cache key $k$ securely, by relying on file system permissions, or encrypted disk partitions. but this is not nice enough, as it will require special system configuration. i don't like having to deal with this extra complexity of double checking that the partition where the keys are cached is encrypted properly.

but i have found an even nicer approach, that does not require any special system configuration. i.e. we can use a normal disk to save the cached keys in, and yet do it in such a way that if an adversary stole the disk, he will have to face the consequences of a too-large entropy, up to a point he wouldn't benefit from stealing user's hard disk.

### 2.3.1 cache's properties

1. each cached key is indexed by a unique $8V$ bits hash.
2. each cached content is $8W$ bits encrypted by another unique $8V$ bits hash. if the clear text data is less than $8W$, clear text is padded until $8W$ is achieved. clear text data cannot be larger than $8W$.
3. insertion, deletion, and retrieval is $O(1)$.
4. brute forcing is $O(n+2^{H(\mathcal{O}_p)})$, where $n$ is total number of cached entries, and $H(\mathcal{O}_p))$ is password's entropy. so, if an adversary wants to figure out password $p$ which gave some cached object, he needs to deal with $H(\mathcal{O}_p) + \log_2 n)$ entropy bits.

### 2.3.2 cache initialisation

1. fill the cache with lots of fake entries. e.g. $n = 2^{30}$ many fake entries.

a fake entry is one that its key is $8V$ bits of cryptographically secure random number, and its content is $8W$ of the same. from an adversary's point of view, there is no distinction between a real cached key entry, and a fake one.

### 2.3.3 cache insertion

1. derive key $k$ by using some key derivation function, such as *ciphart*, *argon2*, etc.
2. let $j$ be the hash of all parameters that were given to the key derivation function. the parameters include user's password, salt, memory requirement, etc.
3. $\hat{j} \leftarrow \texttt{hash}(j\|0, 8V)$.
4. $\hat{k} \leftarrow \texttt{enc}(k, 0, j)$.
5. insert the following entry into the keys cache:
   - entry's key is $\hat{j}$.
   - entry's content is $\hat{v}$.

### 2.3.4 cache retrieval

let's say that the user wants to retrieve the expensively derived key, $k$, from the cache.

1. user calculates $j$ by hashing all parameters of the key derivation function.
2. user calculates $\hat{j} \leftarrow \texttt{hash}(j\|0, 8V)$.
3. user queries cache by asking for entry with key $\hat{j}$.
4. user obtains data associated with key $\hat{j}$, which is $\hat{k}$.
5. user retrieves the key that he wanted, $k$, by calculating $k \leftarrow \texttt{dec}(\hat{k}, 0, j)$.

### 2.3.5 potential adversary strategies

say that the adversary's

- input: the expensively derived key $v$, knows all parameters of the key derivation which gave $v$, including knowledge of the process the generated the password $p$, except that he doesn't know the password $p$. also let's say that the adversary stole user's hard disk, which contains user's keys cache.

- goal: to find $p$ which gave $v$. let's say that the cache was initialised with $2^{30}$ many fake entries. so what can the adversary do?

because the adversary knows the process that generated $p$, he knows $p$'s shannon's entropy $H(\mathcal{O}_p)$. i.e. he knows that $p$ is one of $2^{H(\mathcal{O}_p)}$ many options. the adversary will need to:

1. calculate $2^{H(\mathcal{O}_p)}$ many different instances of $j$, i.e. $j_0, j_1, \ldots, j_{2^{H(\mathcal{O}_p)-1}}$, each calculated with a different password candidate.

2. for any cached data $\hat{v}$ in the $2^{30}$ many entries, try to decrypt it with each key in $j_0, j_1, \ldots, j_{2^{H(\mathcal{O}_p)-1}}$ until $v$ is found.

this means that the brute forcing space of the adversary is $2^{H(\mathcal{O}_p)} 2^{30} = 2^{H(\mathcal{O}_p)+30}$, which, unless `hash` is broken, is not reducible. therefore, by applying lemma 1.1, the adversary needs to answer no less than $H(\mathcal{O}_p) + 30$ many theoretical binary questions.

so, while password $p$'s entropy is just $H(\mathcal{O}_p)$, the adversary needs to suffer a higher entropy when trying to decode the keys cache. specially. the adversary needs to suffer $H(\mathcal{O}_p) + 30$.

effectively, it is easily possible that $H(\mathcal{O}_p) + 30$ is the same amount of entropy that the adversary would need to suffer if he was to trying to find $p$ by brute forcing the key derivation function. i.e. the adversary did not benefit from stealing user's hard disk!

## 2.4 "hashing is only for compression" law

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

- option 1 — use only nails and, when you need screws, modify some nails into screws.

- option 2 — use nails and screws.

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 screws alongside nails, except that the screws are constrained by being re-invented by modifying nails. in other words, option 1 has the extra assumption that its screws must be made using nails, while option 2 does not have this extra assumption. hence option 2 is actually simpler.

my answer with *ciphart* is that option 2 is the simpler choice because it removes the re-invention aspect, specially if the re-invented screws were worse than screws that were made as screws from the start.

on the other hand, *argon2* acts as if option 1 is better, which is wrong at every level.

before i write about how *argon2* is an example of adopting the mistake in option 1, i want to define the nails and the screws of a key derivation function, strictly from the perspective of key derivation functions.

**definition 2.4** (hashing functions as seen by key derivation functions)**.** *a function that maps input to output such that:*

- *unlimited input — input size is an unbounded number of concatenated data chunks.*

- *compression — input's size could be larger than output's size.*

- *preserving entropy is tried — output should have as much entropy bits from the input, but entropy loss is possible, as shannon's entropy of the input could be more bits than output's bits.*

- *walking backwards is extremely hard — analysing the output to find the input is computationally too hard.*

**definition 2.5** (symmetric block encryption functions as seen by key derivation functions)**.** *a function that maps input to output such that:*

- *limited input — fixed sized data and a fixed sized key.*

- *preserving entropy is guaranteed — no entropy loss is possible. the proof is that, if the encryption key is known, we can bring back every input bit from the output by decryption.*

- *walking backwards is extremely hard — analysing the output to find the input is computationally too hard.*

when *argon2* completes solving the tasks in its memory pad, it derives the output key by hashing certain chunks of bytes in the pad. the number of hashed chunks depends on pad's size, so it's not of a fixed size, and henceforth meets the properties of a hashing function shown in definition 2.4. therefore, *argon2* using a hashing function at this stage is justified.

however, when *argon2* is solving tasks in the memory pad, it still uses a hashing function, despite the fact that it is strictly dealing with inputs of a fixed size, which rather meets the properties of a symmetric block encryption function in definition 2.5. here, *argon2* better use a symmetric block encryption function instead of a hashing function. using a hashing function here is a problem due to:

- re-invention of wheels — having *argon2* concatenate two inputs of a fixed size in order to derive an output of the same size is effectively an attempt to reinvent a symmetric block encryption function. i.e. the concatenation is used to emulate the effect of having a pre-shared key which already exists in symmetric block encryption functions. why re-invent keys by concatenation when there exists functions that already have keys?

- needless risk of entropy loss — using a hashing function when dealing with fixed input sizes needlessly increases the probability of having potential entropy losses. this is due to the fact that hashing functions *try* to preserve input's entropy, but cannot guarantee it, while symmetric block ciphers *do* guarantee it[9]. so why have the possibility of losing entropy bits when you don't have to?

- slower memory filling rate — generally speaking, hashing functions tend to be slower than symmetric block encryption functions. this is because of dealing with compression is harder than not.

  symmetric block encryption functions guarantee preserving input's entropy in the output with much less effort thanks to the fact that the output is at least as large as the input.

  but hashing functions don't have the luxury of having an output that's as large as the input, thus they need to work a lot more in order to ensure that no input entropy is needlessly lost.

  this slowness is bad as it reduces number of passes over the memory pad in a unit of time. more passes over the memory pad are important for strengthening the memory hardness.

**law 2.1** (hashing is only for compression)**.** *use hashing functions only when compression happens. otherwise, use symmetric block encryption functions.*

# 3 caveman's keys cache

# 4 ciphart

*ciphart* is basically a variation of *argon2*, except that it uses the fundamental ideas in section 2 to be:

- easier by injecting shannon's entropy bits into its derived keys; thanks to my "perfect lie" theorem (theorem 2.2).

- memory-*harder* by utilising space beyond the random-access memory, by also utilising the hard-disk; thanks to my "cacheable keys" discovery (discovery 2.1).

- faster by using `hash` only when compression takes place, otherwise using `enc`; thanks to my "hashing is only for compression" law (law 2.1).

[9]the proof is that a symmetric block encryption function has a decryption function to bring the original input back from the output, while a hashing function may not have as such.

## 4.1 the algorithm

### 4.1.1 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 shannon's entropy bits to inject into output key. |
| $K$ | output key's size in bytes. |

### 4.1.2 internal variables

| | |
|---|---|
| `enc` | data encryption function with nonce and key. |
| `dec` | data decryption function with nonce and key. |
| `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.

$C \leftarrow \begin{cases} 64 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes} \\ \vdots \end{cases}$ this to reflect the block size of the encryption algorithm that implements `enc`.

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

$R$ $V$ bytes of cryptographically secure random number. this is a temporary key for encrypting and decrypting the $F$ file, that's forgotten upon *ciphart*'s completion. why? so that we can just delete the $F$ file normally, without worrying about having its content remain in the disk.

$J$ $\leftarrow \texttt{hash}(P \| S \| M \| D \| \ldots \| K, V)$. $V$ bytes key to encrypt and decrypt cached key objects

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

$m_i$ $C$-bytes memory for $i^{th}$ task in the $\hat{M}$-bytes pad.

$n_l$ $\leftarrow \max(\text{nonce})L^{-1}l$. nonce variable for $l^{th}$ lane. $n_0L$ is also used as a counter to measure total number of times `enc` was called.

$f$ $\leftarrow 0$. 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.

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

$v$ $\leftarrow J$. $V$ bytes key. $v$ is the key itself, and $*v$ is a pointer to it.

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

#### 4.1.3 output

$k$ $K$ bytes key, with $H(\mathcal{O}_p) + \log_2 n_0L$ many shannon's entropy bits, such that $\log_2 n_0L \geq B$.

#### 4.1.4 steps

steps of *ciphart-d* is shown in algorithm 2, which corresponds to *argon2d*. defining *ciphart-i* or *ciphart-di* variants, which correspond to *argon2i* or *argon2id*, respectively, is a trivial matter; i just didn't bother because i don't need them yet.

### 4.2 noteworthy features

#### 4.2.1 parallelism

iterations of the loop in line 7 in algorithm 2 are independent, so can be done in $L$ cpu cores.

#### 4.2.2 memory-hardness

*Proof.* algorithm 2 is just a variation of *argon2d*. so if *argon2d* is memory-hard, then so is *ciphart*. □

#### 4.2.3 memory-*harder*ness

thanks to discovery 2.1, 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 2.

   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

**algorithm 2:** ciphart-d

1 **if** $Y$ **and** $\texttt{exists}(Z)$ **then**
2 $\quad \hat{k} \leftarrow \texttt{read}(Z, 0, K)$;
3 $\quad k \leftarrow \texttt{dec}(\hat{k}, 0, J)$;
4 $\quad$ **return** $k$
5 **while** 1 **do**
6 $\quad$ **for** $g \leftarrow 0, 1, \dots, G-1$ **do**
7 $\quad\quad$ **for** $l \leftarrow 0, 1, \dots, L-1$ **do**
8 $\quad\quad\quad$ **for** $t \leftarrow 0, 1, \dots, T-1$ **do**
9 $\quad\quad\quad\quad i \leftarrow gLT + lT + t$;
10 $\quad\quad\quad\quad$ **if** $t < T-1$ **then**
11 $\quad\quad\quad\quad\quad j \leftarrow i + 1$;
12 $\quad\quad\quad\quad$ **else if** $t = T-1$ **then**
13 $\quad\quad\quad\quad\quad j \leftarrow i - T + 1$;
14 $\quad\quad\quad\quad m_j \leftarrow \texttt{enc}(m_i, n_l, *v)$;
15 $\quad\quad\quad\quad n_l \leftarrow n_l + 1$;
16 $\quad\quad\quad\quad$ **if** $f = 0$ **then**
17 $\quad\quad\quad\quad\quad *v \leftarrow m_j \bmod (gLTC + tC - V)$;
18 $\quad\quad\quad\quad\quad$ **if** $*v \geq gLTC - V$ **then**
19 $\quad\quad\quad\quad\quad\quad *v \leftarrow *v + lTC$;
20 $\quad\quad\quad\quad$ **else**
21 $\quad\quad\quad\quad\quad *v \leftarrow m_j \bmod (\hat{M} - LTC + tC - V)$;
22 $\quad\quad\quad\quad\quad$ **if** $*v \geq gLTC + tC - V$ **then**
23 $\quad\quad\quad\quad\quad\quad *v \leftarrow *v + LTC$;
24 $\quad\quad\quad\quad\quad$ **else if** $*v \geq gLTC - V$ **then**
25 $\quad\quad\quad\quad\quad\quad *v \leftarrow *v + lTC$;
26 $\quad\quad$ **if** $d \leq D$ **then**
27 $\quad\quad\quad$ **for** $i \leftarrow gLT, \dots, gLT + (T-1)$ **do**
28 $\quad\quad\quad\quad \hat{m}_i \leftarrow \texttt{enc}(m_i, d, R)$;
29 $\quad\quad\quad\quad \texttt{write}(F, \hat{m}_i)$;
30 $\quad\quad\quad\quad d \leftarrow d + 1$;
31 $\quad\quad\quad\quad$ **if** $d \geq D$ **then**
32 $\quad\quad\quad\quad\quad$ **break**;
33 $\quad\quad$ **else**
34 $\quad\quad\quad h \leftarrow h + L$;
35 $\quad\quad\quad$ **if** $h \geq H$ **then**
36 $\quad\quad\quad\quad \hat{d} \leftarrow v \bmod (d - V)$;
37 $\quad\quad\quad\quad \hat{v} \leftarrow \texttt{read}(F, \hat{d}, V)$;
38 $\quad\quad\quad\quad v \leftarrow v \oplus \texttt{dec}(\hat{v}, d, R)$;
39 $\quad\quad\quad\quad u \leftarrow u + 1$;
40 $\quad\quad\quad\quad h \leftarrow 0$;
41 $\quad\quad\quad$ **if** $f \geq 1$ **and** $u \geq 1$ **and** $n_0 L \geq 2^B$ **then**
42 $\quad\quad\quad\quad g_{\text{last}} \leftarrow g$;
43 $\quad\quad\quad\quad$ **go to** line 45;
44 $\quad f \leftarrow f + 1$;
45 $i \leftarrow g_{\text{last}} LT$;
46 $k \leftarrow \texttt{hash}(m_{i+0T} \| m_{i+1T} \| \dots \| m_{i+(L-1)T}, K)$;
47 **if** $Y$ **then**
48 $\quad \hat{k} \leftarrow \texttt{enc}(k, 0, J)$;
49 $\quad \texttt{write}(Z, \hat{k})$;
50 $\texttt{delete}(F)$;
51 **return** $k$

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 35 in algorithm 2.

   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.

# 5 benchmarks

# 6 application scenarios

## 6.1 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 2.2 and discovery 2.1.

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 6.1.** *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.*

## 6.2 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 2.2, this will automatically increase shannon's entropy of all users' passwords without requiring users to memorise harder passwords. thanks to discovery 2.1, 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.

perhaps, one day in the future, the *password* field in *html* should have an argument called *kdf* that goes like `<input type="password" kdf="ciphart" ...>` which specifies the key derivation function that the web browser must use to derive a more secure key, and then send the derived key instead of sending the password. browsers are already kind enough to *manage* our passwords for us, so this will be just a matter of making them send more secure key as opposed to the less secure passwords.

i personally don't let browsers manage my passwords, because i don't trust them, and because i don't need them, as i use my own passwords manager *nsapass*. so i don't care about this. i just wrote it in case it helps the cute normal people more securely syndicate videos of their happy puppies across the interwebs, without requiring them to become better people. why? i guess i'm too kind, and i think these people are sort of adorable the way they are, so i don't mind to preserve their innocence.

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. ideally implemented in a passwords manager like *nsapass* as i said earlier, or integrated into the password fields of web browsers in order to preserve the innocence of the adorable normies.

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

# A donations

no big deal, but i want to see how it feels to be like those people who get donated to. *bryan lunduke*[11] said once, in a very energetic manner, that it feels like being like "people who get donated"[12]. so, i guess it's one of those unexplainable things that need to be tried.

## A.1 bitcoin

`bc1qtylzjtgd0yu4v7f8hyfzufn7nu692v9fc88jln`

[11] `http://lunduke.com`
[12] `https://youtu.be/radmjL5OIaA?t=12`