ciphart

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

caveman[1]

2021-03-12 01:17:10+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 "entropy injection" 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 (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 $o \in \{o_0, o_1, \ldots, o_{10-1}\}$? 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 $o \in \{o_0, o_1, \ldots, o_{5-1}\}$? 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.

shannon's entropy tells us the minimum number of questions to be asked 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 options 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 a 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 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 trees.

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 $2^{H(\mathcal{O}_p)}$ many password candidates.

lemma 1 (walking backwards to entropy)**.** *if, asymptotically on average, the minimum number of candidate password tests to find a key that matches p's key is x many tests, then we know that $H(\mathcal{O}_p) = \log_2 x$.*

1.2 passwords' security

definition 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 a $8V$ bit key $k \leftarrow \texttt{hash}(p, 8V)$, derived from password p. say that the attacker 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:

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

one way to estimate the `cost` function 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]. the housekeeping of this approach is expensive, and is usually not possible to get any guarantees as we don't know

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

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 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 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:

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

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

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

(4) becomes:

$$\begin{aligned}2^{H(\mathcal{O}_p)}(N+0) &= N2^{H(\mathcal{O}_p)}\\ &= 2^{H(\mathcal{O}_p)+\log_2 N}\end{aligned} \tag{7}$$

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 $e(0) = 1$.

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

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

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

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

(4) becomes:

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

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 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:*

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

2 fundamental ideas

2.1 "entropy injection" theorem

the information content of a thing x, also known as its entropy $H(\mathcal{O}_x)$, is the minimum number of binary questions that, if answered, x would be found. in other words, for any $o \in \mathcal{O}$, we will know the answer of "is $o = x$?".

entropy originates from randomness. e.g. password p has some entropy because the process that generated it contain some randomness. an example of such process is a process that uniformly and randomly samples, with replacement, say, 5 many words from a bag containing, say, 100 words. in this example, adversary's options of what p might be, given his knowledge of this randomised process, is in the options set $\mathcal{O}_p$, which contains all possible combinations of 5 words, with cardinality $|\mathcal{O}_p| = 100^5$. since this process is using uniform random distribution to sample the words, $H(\mathcal{O}_p) = \log_2 100^5 = 33.22$. meaning, asymptotically on average, the adversary has to ask, at least, 33.22 binary questions.

the key to inject entropy into a thing, is a matter of defining a process that forces the adversary to ask more questions. in the example above, the process used randomness to force the adversary to ask questions about its sampled words.

the key point of this section is to answer the question: can cryptographic functions be used to force the adversary

to ask more questions about a thing? to which my answer is "yes, we can", and my proof will show it. but first, we have to define what a question means in our context.

2.1.1 what is a question

definition 4 (a question in password brute-forcing)**.** *if p is an unknown password that the adversary got its key $k \leftarrow \texttt{hash}(p, 8V)$, then a question is: whether a candidate password $\hat{p}$ satisfies $\texttt{hash}(\hat{p}, 8V) = k$. which means that a question is comprised of two things only:*

1. *calculating a hash,*
2. *and then testing an equality.*

if the adversary is forced to do (1) and (2), in order, then he has asked a question, and got its answer.

note 1. *this question in definition 4 is different than the theoretical binary question given by shannon's entropy. the question in definition 4 is what causes the cardinality $|\mathcal{O}_k|$ to increase, while the theoretical binary question of shannon's entropy is the $\log_2$ of this cardinality.*

in other words, question asking and answering is very physical, as it requires measurable physical changes, such as executing cpu instructions. when the adversary does (1) and (2), in order, then he has asked a question.

if the adversary did (1) and (2), he cannot reject that he asked a question by simply saying that he didn't have the *intention* to ask. his intentions are irrelevant. the fact remains that he *physically did* (1) and (2), and that *is* the *act* of asking a question and getting its answer.

so, in order to force the adversary to ask more questions, all we have to do is to force him to *physically* do more of (1) and (2). once he does the act, he asked the questions and got their answers.

2.1.2 an algorithm that forces questions

say that p is a password that all ambiguities about it are resolved after getting, on average, $H(\mathcal{O}_p)$ many balanced binary questions, $q_0, q_1, \ldots, q_{H(\mathcal{O}_p)-1}$, answered.

case 1 — say that the adversary got p's hash, $k \leftarrow \texttt{hash}(p\|s, 8V)$, and that his goal is to find out p by brute-forcing. in this case, the adversary will be considering password candidates $p_0, p, \ldots, p_{2^{H(\mathcal{O}_p)-1}}$, and tests them by asking the following binary questions in order to guarantee finding p:

- q_0: is $\texttt{hash}(p_0\|s, 8V) = k$?
- q_1: is $\texttt{hash}(p_1\|s, 8V) = k$?

  $\vdots$
- $q_{H(\mathcal{O}_p)-1}$: is $\texttt{hash}(p_{H(\mathcal{O}_p)-1}\|s, 8V) = k$?

case 2 — say that `mhash`,was used instead of just `hash`; i.e. the adversary got $\hat{k} \leftarrow \texttt{rhash}(p\|s, 8V, N)$ instead. in this case, the adversary will seek to answer the following binary questions:

- $\hat{q}_0$: is $\texttt{rhash}(p_0\|s, 8V, N) = k$?
- $\hat{q}_1$: is $\texttt{rhash}(p_1\|s, 8V, N) = k$?

  $\vdots$
- $\hat{q}_{H(\mathcal{O}_p)-1}$: is $\texttt{rhash}(p_{H(\mathcal{O}_p)-1}\|s, 8V, N) = k$?

so, you may say that we're still in need to ask, on average, $H(\mathcal{O}_p)$ many balanced binary questions, right? i disagree.

i think what is happening with `rhash` is that each question $\hat{q}_0, \hat{q}_1, \ldots \hat{q}_{H(\mathcal{O}_p)-1}$ is a *bundled* question, each, containing N many balanced binary questions.

i.e., for any question $\hat{q}_i$, answering it is equivalent to answering the N-deep recursion over `hash`:

$\hat{q}_i$: is $\texttt{hash}(\texttt{hash}(\ldots\texttt{hash}(p_i\|s, 8V), \ldots, 8V), 8V) = k$?

which its answer is "yes" if and only if all of the following N many questions are answered by "yes":

- is $\texttt{hash}(p_i\|s, 8V) = k_0$?
- is $\texttt{hash}(k_0\|s, 8V) = k_1$?
- is $\texttt{hash}(k_1\|s, 8V) = k_2$?

  $\vdots$
- is $\texttt{hash}(k_{N-1}\|s, 8V) = k$?

meaning, if any $\hat{q}_i$ question is answered, it necessarily means that we have answered N many q_i-like questions. if answer to $\hat{q}_i$ is "yes", it means that the answer to each of those N many q_i-like questions is also "yes". if answer to $\hat{q}_i$ is "no", it also means that the answer to each of those N many questions is also "no"[8].

in other words, functions `rhash` and `mhash` are using cryptography to force the adversary to need to answer more questions beyond $H(\mathcal{O}_p)$. `rhash` increases questions by a factor of N. `mhash` increases questions by a factor of $N^{e(M-A)}$.

theorem 1 (entropy injection)**.** *for any password p, and any positive numbers V, N and M:*

$$H\Big(\texttt{rhash}(p\|s, 8V, N)\Big) = C\Big(p, \texttt{rhash}(\ldots, N)\Big)$$

$$H\Big(\texttt{mhash}(p\|s, 8V, N, M)\Big) = C\Big(p, \texttt{mhash}(\ldots, N, M)\Big)$$

2.2 "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.*

[8] unless hashing collisions happen, which is very unlikely.

2.2.1 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.

2.2.2 potential adversary strategies

let's see what may the adversary 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.

2.3 "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, when *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 5 (hashing functions as seen by key derivation functions)**.** *a function that maps input to output such that:*

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

- *compression — input's shannon's entropy bits could be larger than the total bit size of the output.*

- *preserving entropy is tried — output should have as much entropy bits from the input, but entropy loss is possible.*

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

definition 6 (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 5. therefore, *argon2* using a hashing function at this stage is justified.

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 fixed lengths, which also meets the properties of a symmetric block encryption function in definition 6. 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 re-invent a symmetric block encryption function. i.e. the concatenation is used to emulate the effect of having a pre-shared key which 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. 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 ciphers 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 1 (hashing is only for compression)**.** *use hashing functions only when compression happens. otherwise, use symmetric block encryption functions.*

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

$V \leftarrow \begin{cases} 32 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{xchacha20} \\ 16 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-128} \\ 32 \text{ bytes} & \text{if } \texttt{enc} \text{ is } \textit{aes-256} \\ \vdots \end{cases}$
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.

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

$n_l \leftarrow \max(\text{nonce})L^{-1}l$. nonce variable for l^{th} lane.

$f \leftarrow 0$. 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.

$*v \leftarrow *\texttt{hash}(P\|S\|M\|D\|\ldots\|K, V)$. 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 $\log_2 n \geq B$.

$\hat{M}$ actual number of bytes required to exist in random-access memory.

d actual number of bytes required to exist in hard-disk.

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, d ← read(Z, 0, V + sizeof(k‖n‖d));
3      return k, n, d
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                 u ← u + 1;
37                 h ← 0;
38             if f ≥ 1 and u ≥ 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, k‖n‖d);
46 delete(F);
47 return k, n, d
```

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*. 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[9] *— 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.*

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.

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.

A donations

this work is sponsored by an unexpected nerdiness. nothing here was supposed to happen. the cause remains unknown, so we call it *something random with a large entropy.*

not-so-ancient mythology[10] 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`

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

[10] i made this myth up.