key derivation with easier measurable security

caveman

January 2, 2021

hi — i propose *ciphart*, a sequential memory-hard key derivation function that has a security gain that's measurable more objectively and more conveniently than anything in class known to date.

to nail this goal, *ciphart*'s security gain is measured in the unit of *relative entropy bits.* relative to what? relative to the encryption algorithm that's used later on. therefore, this *relative entropy bits* measure is guaranteed to be true when the encryption algorithm that's used with *ciphart* is also the same one that's used to encrypt the data afterwards.

my reference implementation is available here[1].

content

1 ciphart **1**

2 parallelism **2**

3 sequential-memory hardness **2**

3.1 example 2

4 security interpretation **2**

5 comparison **3**

6 summary **3**

1 ciphart

parameters:

M	each task's size, at least 32 bytes.
W	total memory in multiples of $2M$.
R	number of rounds per task.
B	added security in *relative entropy bits.*
enc	encryption function.
k	initial key.

input:

T	$\leftarrow W/M$
P	$\leftarrow \max(2, \lceil 2^B/(TR) \rceil)$
x	$\leftarrow 0$, a 16 bytes wide variable.
m_t	for any task $t \in \{1, 2, \ldots, T\}$, m_t is M-bytes memory for t^{th} task to work on. $m_t[0:16]$ means first 16 bytes. $m_t[-16:]$ means last 16 bytes.
nonce	a variable with enough bytes to store nonces in.
hash	a function to compress W bytes into desired key length.

output:

$\hat{k}$ better key, with B, or more, *relative entropy bits.* specifically, with $\log_2(PTR) \geq B$ bits.

steps:

for $p = 1, 2, \ldots, P$ **do**
  for $t = 1, 3, \ldots, T-1$, in steps of 2 **do**
    $i \leftarrow t$
    $j \leftarrow t + 1$
    for $r = 1, 2, \ldots, 2R$ **do**
      nonce $\leftarrow (p, t, r)$
      $m_i \leftarrow \text{enc}(m_j, \text{nonce}, k)$
      $\hat{i} \leftarrow i$
      $i \leftarrow j$
      $j \leftarrow \hat{i}$
    end for
    $x \leftarrow x \oplus m_i[-16:]$
    $x \leftarrow x \oplus m_j[-16:]$
  end for
  for $t = 1, 2, \ldots, T$ **do**
    $m_t[0:16] \leftarrow m_t[0:16] \oplus x$
  end for
end for
return $\hat{k} \leftarrow \text{hash}(m_1, m_2, \ldots, m_T)$

[1] `https://github.com/Al-Caveman/ciphart`

2 parallelism

iterations inside the `for` loop, in step 2, are independent of one another, so we can distribute them happily across different threads to achieve maximum cpu utilisation.

exception is in steps 12 and 13, where a mutex might be required as the xor assignment there is not necessarily atomic. but this is not a real problem since the real expensive part is in step 7, which overshadows the overhead of the mutexes.

plus, even if one has an ultra-fancy hardware where step 7 is so lightweight that it effectively competes with the mutexes needed for steps 12 and 13, then one can simply increase number of rounds R until the correct order is restored.

3 sequential-memory hardness

all of the tasks solved in the first pad, i.e. when $p = 1$, can be normally computed sequentially with just, say, $2M$ bytes memory.

sequential-memory hardness is introduced for later pads, $p \geq 2$, as the x variable gets applied to tasks' memory workspaces $m_1, m_2, \dots, m_T$, which causes every task's memory in pad p to depend on every task's last 16 bytes of the previous pads $p-1, p-2, \dots, 1$.

3.1 example

say that we've got a total memory of $W = 10 \times 2M$ bytes, which basically means we have 10 pairs of tasks (or $T = 20$ tasks) in a pad.

also say that we've got B large enough that caused us to need 5 pads, i.e $P = \max(2, \lceil 2^B/(TR) \rceil) = 5$.

for simplicity, let's say $R = 1$. also, say that the adversary has enough memory to keep track of x values across the different pads. since each x is only 16 bytes, and since we've got 5 pads, this means that the adversary has managed 16×5 bytes to not worry about losing $x$es across the pads.

the question of this example is: how many times will the function enc be called if we wanted to complete the ciphart algorithm with just $W = 2M$ bytes for the pad instead of $W + 10 \times 2M$ bytes?

1. in the first pad, $p = 1$, enc will be called 20 times. by the end of it, we will x populated with all xor-ed data from all the 20 tasks, and end up having the content of the two tasks, say m_{19} and m_{20}. but we won't have the content of the other tasks, since we said we have only $2M$ bytes pad size limitation in this example.

2. in the second pad, $p = 2$, we will be able to start working on tasks $t = 19$ and $t = 20$ since we already have their memory content m_{19} and m_{20} as well as the xor-ed data x. so with these two tasks, life is easy. but what about other tasks, say:

   (a) tasks $t = 1$ and $t = 2$? we already have x, but we lack their memory content from the previous pad. so we've got to repeat that. meaning enc will be called 2 times from the previous pad, and then 2 times for the current pad $p = 2$. totalling 4 calls.

   (b) tasks $t = 3$ and $t = 4$? same as before, 4. this repeats to the rest of task pairs (except the lucky tasks $t = 19$ and $t = 20$).

   meaning, we've got $20 - 2$ unlucky tasks, each of which will result in calling enc two times in order to solve pad $p = 2$ and obtain its xor-data x.

   in total, the second pad will call enc function $2 + (20 - 2) \times 2 = 38$ times.

3. in the third pad, $p = 3$, the same will repeat, except for calling enc one more time with the unlucky tasks. i.e. $2 + (20 - 2) \times 3 = 56$.

4. in the forth pad, $p = 4$, $2 + (20 - 2) \times 4 = 74$ times.

5. in the fifth pad, $p = 5$, $2 + (20 - 2) \times 5 = 92$ times.

in total, a $W = 2M$ bytes implementation would end up calling enc $20 + 38 + 56 + 74 + 92 = 280$ times. **correction:** i think it will be $20 \times 5 + (20 - 2) \times 5 = 190$.

a $W = 10 \times 2M$ bytes implementation will call enc $20 \times 5 = 100$ times instead.

so.. is ciphart really hard? it is not. increase in calculation is linear (not exponential). this is a failure. there needs to be an algorithmic change.

4 security interpretation

it sucks. don't use it yet. i'm thinking how to fix this garbage.

5 comparison

6 summary