key derivation with easier measurable security

caveman

January 4, 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]`https://github.com/Al-Caveman/ciphart`

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
  end for
  for $t = 1, 2, \ldots, T$ do
    for $\hat{t} = 1, \ldots, t-1$ and $\hat{t} = t+1, \ldots, T$ do
      $m_t[-16:] \leftarrow m_t[-16:] \lll 1$
      $m_t[-16:] \leftarrow m_t[-16:] \oplus m_{\hat{t}}[0:16]$
    end for
  end for
end for
return $\hat{k} \leftarrow \text{hash}(m_1, m_2, \ldots, m_T)$
```

2 parallelism

iterations in steps 2 to 12, are independent of one another, so we can distribute them happily across different threads to achieve maximum cpu utilisation.

iterations in steps 13 to 18 can also be done in parallel after completion of steps 2 to 12.

3 sequential-memory hardness

steps 13 to 18 is the part where sequential memory-hardness is expected to be born. proof maybe soon. but first i have to actually test it in code to see how fast/slow is it. right now i fear that memory's I/O might become a significant bottleneck.

4 security interpretation

5 comparison

6 summary