CENTRIFUGE

A password hashing algorithm

Rafael Alvarez
ralvarez@dccia.ua.es

Dpt. of Computer Science and Artificial Intelligence
(DCCIA)

University of Alicante
Spain

Specification

Introduction

Centrifuge is a password hashing algorithm based on several key concepts: a block cipher employed as a pseudorandom number generator, a hash function to seed the generator, a large table of pseudorandom sequences that is combined to form the resulting password hash and an evolving substitution box that adds complexity and prevents parallelization.

The interface is configurable in byte increments, accepting variable length password and salt inputs, as well as producing a variable length output. It is also extremely configurable in terms of computational and memory costs, substantially increasing the potential lifespan of the algorithm.

Algorithm

Parameters

Centrifuge has the following internal parameters:

Name	**Type**	**Description**
password	uint8_t *	An array of bytes containing the password
passlen	uint32_t	Length of the password in bytes
salt	uint8_t *	An array of bytes containing the salt
saltlen	uint32_t	Length of the salt in bytes
out	uint8_t *	Buffer to contain the output hash
outlen	uint32_t	Length of the output in bytes
p_mem	uint64_t	Num. of entries in table (memory factor)
p_time	uint64_t	Num. of s-box swaps per entry (time factor)

These parameters are mapped to competition parameters as follows:

```
int PHS(void *out, size_t outlen,
        const void *in, size_t inlen,
        const void *salt, size_t saltlen,
        unsigned int t_cost, unsigned int m_cost){

        uint64_t m,t;

        if((m_cost > 63)||(t_cost > 63)) return -1;

        m = (uint64_t) 1<<(m_cost);
        t = (uint64_t) 1<<(t_cost);
        return cfuge(in,inlen,salt,saltlen,out,outlen,m,t);
}
```

That is, the competition cost parameters are interpreted as the exponents in the corresponding powers of 2, allowing a more meaningful scale. The internal parameters can be used directly if a different scale is required. The parameters must fit in a 64 bit integer, so *t_cost* and *m_cost* must be smaller than 64.

Elements

Algorithm elements are as follows:

H()	A cryptographic hash function that accepts a variable input and generates a 512 bit hash. In centrifuge, the SHA-512 function is used by default.
C()	A cipher function comprised of a block cipher in CFB mode. In centrifuge, the AES cipher with a 256 bit key is used by default.
M[]	An array of p_mem * outlen bytes. This is generated in the build table stage and constitutes the main memory cost of the algorithm.
S[]	An 8x8 substitution box table (array of 256 bytes). The s-box is initialized at the beginning and then evolves continuously, helping to prevent parallelizability. The number of evolutions (swaps) per entry in M constitutes the main time cost of the algorithm.
Seq[]	An array of p_time bytes that is used to evolve S. It is generated using C().

Seeding

The seeding stage takes the password and salt and generates suitable values for the initialization vector (*IV*) and key parameters of *C()*. It also initializes the output buffer.

It computes the hash for the password and for the salt separately and then computes the hash of the concatenated results.

```
uint8_t seedin[128],seedout[64];

H(password,passlen,seedin);
H(salt,saltlen,seedin+64);
H(seedin,128,seedout);
```

Then, the seed is divided in different parts: bytes 0 to 15 are repeated as necessary to initialize the output buffer, and *C()* is initialized with bytes 16 to 31 as the *IV* (128 bits) and bytes 32 to 63 as the key.

```
for(int i=0; i<outlen; i++)
        out[i] = seedout[i%16];

memcpy(iv,seedout+16,16);
memcpy(key,seedout+32,32);

initC(key,iv);
```

The seeding stage is designed to support any hash function that produces a 512 bit output and any block cipher with a 256 bit key and 128 bit block. The seeding scheme could accommodate different parameters if necessary with little change.

Substitution Box Initialization

The s-box *S* is initially setup with values 0 to 255 and then evolved using a pseudorandom buffer (*buf*) of 256 bytes that dictates which values are swapped.

```
uint8_t buf[256];
uint8_t m,l,t;

memset(buf,0,256);
C(buf,buf,256);
for(int i=0; i<256; i++) {
        S[i] = (uint8_t)i;
}

for(int i=0; i<256; i++) {
        m = (uint8_t)i;
        l = buf[i];
        t = S[m];
        S[m] = S[l];
        S[l] = t;
}
```

Table Build

Once the initial state for *S* is established, then the construction of *M* begins. Once *M* is constructed it contains *p_mem* pseudorandom vectors of *outlen* bytes. The process is, therefore, iterated for each entry in table *M* (*p_mem*).

```
uint8_t m,l,t;          // indexes to S
uint64_t offs = 0;      // offset into M

for(uint64_t i=0; i<p_mem; i++) {
```

A pseudorandom buffer of *p_time* bytes (*Seq*) is used to mutate *S* as in the previous step.

```
        C(Seq,Seq,p_time);    // generate sequence
        for(uint64_t j=0; j<p_time; j++) {   // modify S
                m = (uint8_t) j % 256;
                l = Seq[j];
                t = S[m];
                S[m] = S[l];
                S[l] = t;
        }
```

Then the output buffer is processed with *S*, substituting each byte by the corresponding one in *S*.

```
        for(uint32_t j=0; j<outlen; j++) // process output
                out[j] = S[out[j]];
```

Finally, the output buffer is encrypted with *C()* and copied to the corresponding place in *M*.

```
        C(out,out,outlen);            // encrypt output
        memcpy(M+offs,out,outlen);    // copy to M
        offs += outlen;
}
```

Output

The output stage combines values from table *M* obtaining a single output vector of outlen bytes in the process. As in the case of the build stage, the process is iterated *p_mem* times.

```
uint64_t index = 0;   // index into M
uint8_t *ptr;         // pointer to start of current M row
uint8_t m,l,t;        // indexes to S

for(uint64_t i=0; i<p_mem; i++) {    // process entry
```

As in previous steps, *S* is modified swapping elements indexed by a pseudorandom buffer (*Seq*).

```
        C(Seq,Seq,p_time);    // generate sequence

        for(uint64_t j=0; j<p_time; j++) {   // modify S
                m = (uint8_t) j % 256;
                l = Seq[j];
                t = S[m];
                S[m] = S[l];
                S[l] = t;
        }
```

Then, a 64 bit index is generated using *C()* and the corresponding address to *M* stored in *ptr*. This index is pseudorandom so that the memory access pattern cannot be predetermined.

```
        C(&index,&index,8); // generate next index
        ptr = M + (index % p_mem) * outlen; // address to M
```

Then the corresponding vector of *M* is combined with the current state of the output buffer. The process involves processing the output with *S* and the bytewise addition modulo 256 with the indexed vector in *M*.

```
        for(uint32_t j=0; j<outlen; j++)  // process and encrypt output
                out[j] = (uint8_t)(S[out[j]] + ptr[j]);
```

Finally, the output buffer is encrypted at the end of the iteration.

```
        C(out,out,outlen);

}
```

The final output is the state of the output buffer when all iterations are ended.

Statement

The author declares that there are no deliberately hidden weaknesses, backdoors or any ill intent in the design of the proposed algorithm.

The scheme is and will remain available worldwide on a royalty free basis, and the designer is unaware of any patent or patent application that covers the use or implementation of the submitted algorithm.

Security Analysis

Centrifuge is primarily meant to be employed as a password hashing algorithm, for password validation and storage. It can also be used as a key stretching / adapting algorithm to convert a variable length password to a fixed length key with very good avalanche properties. Other applications include the proof-of-work scheme in crypto-coins or anywhere a costly hash function is required.

The security of this kind of algorithm is bound by the complexity of the input password. It is important to stress that properly random and long enough passwords must be used. Ideally, the length for the password and salt inputs should be 32 bytes (256 bit) or longer (smaller is also supported if necessary).

The seed of the algorithm is the result of the *SHA-512* bit hash function. Furthermore, *AES-256* is used in CFB mode as the pseudorandom generator function in the rest of the algorithm. Centrifuge should be on par with this primitives at a minimum security level of 2^{256} for brute force attacks. Specific design decisions have been taken to prevent significant speedups in hardware and dedicated platforms (see performance analysis).

In the case *SHA-512* or *AES-256* were broken or found unsuitable for any reason, centrifuge can employ other primitives with the same interface without any modification. Furthermore, adapting the algorithm to primitives with different interfaces is very simple too.

Performance Analysis

The algorithm has two performance related parameters: *p_mem* and *p_time*. These are independent but related in a subtle way.

The memory parameter determines the number of entries in *M* and, together with the output length, determine the amount of memory employed for *M* (*p_mem * outlen* bytes). It also has an impact on the time aspect, since a bigger table will take longer to generate (build stage) and combine (output stage). This happens too, to a lesser extent, with the output length.

The time parameter determines the number of swaps in *S* for each entry in *M* (build and output stages), therefore impacting the total computation time significantly. It also affect the amount of memory accesses per entry in *M*, which impacts the total memory bandwidth consumption; and the sequence used to evolve *S* requires a small amount of memory too.

There are no hardware (GPU, FPGA, ASIC, etc.) implementations of centrifuge available at the moment of submission. Nevertheless, several steps have been taken in order to prevent significant speedup on these platforms. AES is employed in cipher feedback mode (CFB), this is so the sequence generation cannot be parallelized; also, AES is accelerated in hardware in most current CPUs, bridging the performance gap with dedicated hardware. Besides, the constant evolution of *S* and the fact that the output of each iteration in the build and output stages is processed with *S* and fed back to the cipher function strengthens the sequential nature of centrifuge. Moreover, it can be configured to employ an enormous amount of RAM so that multiple hardware units in parallel are not cost-effective.

Regarding parameter tuning, every application is different and requires specific parameters. A good place to start is *m_cost=16*, *t_cost=8* and *outlen=32*. This consumes 2MB of RAM, transforms *S* completely on every iteration (256 swaps) and takes about a third of a second on a current laptop.

The following measurements have been taken with a 2013 laptop with 16GB of RAM, Core i5 2.6GHz and version 1.0.1f of the OpenSSL crypto library for an output length of 32 bytes.

```
m=00, t=00, 0.000032 sec
m=00, t=01, 0.000008 sec
m=00, t=02, 0.000008 sec
m=00, t=03, 0.000008 sec
m=00, t=04, 0.000009 sec
m=00, t=05, 0.000008 sec
m=00, t=06, 0.000009 sec
m=00, t=07, 0.000012 sec
m=00, t=08, 0.000013 sec
m=01, t=00, 0.000009 sec
m=01, t=01, 0.000010 sec
m=01, t=02, 0.000008 sec
m=01, t=03, 0.000009 sec
m=01, t=04, 0.000010 sec
m=01, t=05, 0.000010 sec
m=01, t=06, 0.000012 sec
m=01, t=07, 0.000015 sec
m=01, t=08, 0.000020 sec
m=02, t=00, 0.000010 sec
m=02, t=01, 0.000010 sec
m=02, t=02, 0.000012 sec
m=02, t=03, 0.000012 sec
m=02, t=04, 0.000012 sec
m=02, t=05, 0.000014 sec
m=02, t=06, 0.000017 sec
m=02, t=07, 0.000023 sec
m=02, t=08, 0.000035 sec
m=03, t=00, 0.000015 sec
m=03, t=01, 0.000016 sec
m=03, t=02, 0.000015 sec
m=03, t=03, 0.000016 sec
m=03, t=04, 0.000017 sec
m=03, t=05, 0.000020 sec
m=03, t=06, 0.000027 sec
m=03, t=07, 0.000039 sec
m=03, t=08, 0.000061 sec
m=04, t=00, 0.000023 sec
m=04, t=01, 0.000023 sec
m=04, t=02, 0.000022 sec
m=04, t=03, 0.000022 sec
m=04, t=04, 0.000027 sec
m=04, t=05, 0.000033 sec
m=04, t=06, 0.000044 sec
m=04, t=07, 0.000067 sec
m=04, t=08, 0.000115 sec
m=05, t=00, 0.000049 sec
m=05, t=01, 0.000034 sec
m=05, t=02, 0.000035 sec
m=05, t=03, 0.000037 sec
m=05, t=04, 0.000042 sec
m=05, t=05, 0.000053 sec
m=05, t=06, 0.000085 sec
m=05, t=07, 0.000121 sec
m=05, t=08, 0.000205 sec
m=06, t=00, 0.000058 sec
m=06, t=01, 0.000060 sec
m=06, t=02, 0.000062 sec
m=06, t=03, 0.000068 sec
m=06, t=04, 0.000086 sec
m=06, t=05, 0.000093 sec
m=06, t=06, 0.000132 sec
m=06, t=07, 0.000216 sec
m=06, t=08, 0.000376 sec
m=07, t=00, 0.000104 sec
m=07, t=01, 0.000103 sec
m=07, t=02, 0.000107 sec
m=07, t=03, 0.000119 sec
m=07, t=04, 0.000138 sec
m=07, t=05, 0.000177 sec
m=07, t=06, 0.000267 sec
m=07, t=07, 0.000438 sec
m=07, t=08, 0.000755 sec
m=08, t=00, 0.000203 sec
m=08, t=01, 0.000200 sec
m=08, t=02, 0.000210 sec
m=08, t=03, 0.000229 sec
m=08, t=04, 0.000269 sec
m=08, t=05, 0.000348 sec
m=08, t=06, 0.000512 sec
m=08, t=07, 0.000852 sec
m=08, t=08, 0.001833 sec
m=09, t=00, 0.000606 sec
m=09, t=01, 0.000510 sec
m=09, t=02, 0.000430 sec
m=09, t=03, 0.000540 sec
m=09, t=04, 0.000720 sec
m=09, t=05, 0.000624 sec
m=09, t=06, 0.000914 sec
m=09, t=07, 0.001494 sec
m=09, t=08, 0.002663 sec
m=10, t=00, 0.000774 sec
m=10, t=01, 0.000750 sec
m=10, t=02, 0.000765 sec
m=10, t=03, 0.000844 sec
m=10, t=04, 0.000986 sec
m=10, t=05, 0.001287 sec
m=10, t=06, 0.001834 sec
m=10, t=07, 0.003067 sec
m=10, t=08, 0.005455 sec
m=11, t=00, 0.001404 sec
m=11, t=01, 0.001399 sec
m=11, t=02, 0.001466 sec
m=11, t=03, 0.001615 sec
m=11, t=04, 0.001902 sec
m=11, t=05, 0.002470 sec
m=11, t=06, 0.003640 sec
m=11, t=07, 0.006140 sec
m=11, t=08, 0.010956 sec
m=12, t=00, 0.002796 sec
m=12, t=01, 0.002794 sec
m=12, t=02, 0.002928 sec
m=12, t=03, 0.003222 sec
m=12, t=04, 0.003801 sec
m=12, t=05, 0.004939 sec
m=12, t=06, 0.007272 sec
m=12, t=07, 0.013166 sec
m=12, t=08, 0.022907 sec
m=13, t=00, 0.005643 sec
m=13, t=01, 0.005582 sec
m=13, t=02, 0.005861 sec
m=13, t=03, 0.006449 sec
m=13, t=04, 0.007624 sec
m=13, t=05, 0.009882 sec
m=13, t=06, 0.014892 sec
m=13, t=07, 0.024836 sec
m=13, t=08, 0.043120 sec
m=14, t=00, 0.011182 sec
m=14, t=01, 0.011250 sec
m=14, t=02, 0.011768 sec
m=14, t=03, 0.012944 sec
m=14, t=04, 0.015248 sec
m=14, t=05, 0.019795 sec
m=14, t=06, 0.029563 sec
m=14, t=07, 0.049733 sec
m=14, t=08, 0.087571 sec
m=15, t=00, 0.022450 sec
m=15, t=01, 0.022511 sec
m=15, t=02, 0.023587 sec
m=15, t=03, 0.025954 sec
m=15, t=04, 0.030553 sec
m=15, t=05, 0.039651 sec
m=15, t=06, 0.059654 sec
m=15, t=07, 0.099596 sec
m=15, t=08, 0.174251 sec
m=16, t=00, 0.045163 sec
m=16, t=01, 0.045061 sec
m=16, t=02, 0.047230 sec
m=16, t=03, 0.051981 sec
m=16, t=04, 0.061301 sec
m=16, t=05, 0.079903 sec
m=16, t=06, 0.119504 sec
m=16, t=07, 0.198972 sec
m=16, t=08, 0.346891 sec
m=17, t=00, 0.092798 sec
m=17, t=01, 0.092692 sec
m=17, t=02, 0.097049 sec
m=17, t=03, 0.106836 sec
m=17, t=04, 0.135823 sec
m=17, t=05, 0.162166 sec
m=17, t=06, 0.244391 sec
m=17, t=07, 0.396109 sec
m=17, t=08, 0.704168 sec
m=18, t=00, 0.191867 sec
m=18, t=01, 0.193146 sec
m=18, t=02, 0.203228 sec
m=18, t=03, 0.220671 sec
m=18, t=04, 0.257448 sec
m=18, t=05, 0.333061 sec
m=18, t=06, 0.488976 sec
m=18, t=07, 0.798545 sec
m=18, t=08, 1.402967 sec
m=19, t=00, 0.393301 sec
m=19, t=01, 0.395757 sec
m=19, t=02, 0.413250 sec
m=19, t=03, 0.454045 sec
m=19, t=04, 0.527622 sec
m=19, t=05, 0.668995 sec
m=19, t=06, 0.989219 sec
m=19, t=07, 1.608019 sec
m=19, t=08, 2.865490 sec
m=20, t=00, 0.794538 sec
m=20, t=01, 0.801910 sec
m=20, t=02, 0.835682 sec
m=20, t=03, 0.911056 sec
m=20, t=04, 1.057780 sec
m=20, t=05, 1.346517 sec
m=20, t=06, 1.985173 sec
m=20, t=07, 3.310282 sec
m=20, t=08, 5.639092 sec
```