

The Pufferfish Password Hashing Scheme

Jeremi M. Gosney
jgosney@stricture-group.com

Version 1.1
January 31, 2015
Seattle, Washington

Abstract

I present *Pufferfish*, a password hashing scheme (PHS) based upon a derivative of the Blowfish ^[1] block cipher and bcrypt's ^[2] Eksblowfish ^[3] algorithm. It may also be used as a key derivation function (KDF).

Bcrypt has long been regarded by many as the de facto standard in password security – arguably the most touted answer to the “Which algorithm should I use to hash passwords?” question, especially within the password cracking community. While bcrypt has withstood the test of time for the past 15 years, it also has its share of shortcomings which may prevent it from being a viable option for the next 15 years: it does not take advantage of modern 64-bit processors; it uses a small, fixed amount of memory; it runs about the same speed on Central Processing Units (CPUs) as it does on Graphics Processing Units (GPUs); it has a maximum input of 72 bytes; and it produces fixed-length output.

Pufferfish attempts to address these issues while increasing resistance to acceleration by re-writing the Blowfish block cipher to use 64-bit words and dynamic, arbitrarily-sized, password-dependent S-boxes while retaining most of the construction of the Eksblowfish algorithm. It has been designed to be faster than bcrypt on CPUs at comparable settings, while being slower than bcrypt on GPUs. It also supports variable-length output so that it may be used to derive an arbitrary number of bits as a KDF.

Acknowledgements

Pufferfish builds upon prior works by Bruce Schneier (Blowfish), and Niels Provos & David Mazieres (bcrypt). However, Pufferfish is not endorsed by these individuals. Its design was inspired by work to accelerate bcrypt cracking by Steve Thomas, Jens Steube, Alexander Peslyak, Sayantan Datta, and Katja Malvoni. A special thanks is also due to Steve Thomas providing assistance and reviewing the reference implementation.

The Algorithm

The following pseudo code attempts to illustrate the Pufferfish algorithm:

Parameter *t_cost*: Time cost, log2 iteration count
Parameter *m_cost*: Memory cost, log2 kibibytes
Parameter *outlen*: Length of output in bytes
Input *pwd*: The password
Input *salt*: A salt
Output *output*: The password-derived *outlen*-long key
P-array: Initialized with first 144 digits of Pi (18 64-bit words)

```
function pufferfish (pwd, salt, t_cost, m_cost, outlen)
    sbbox_words  $\leftarrow 2^{(m\_cost + 5)}$ 
    salt_hash  $\leftarrow \text{hmac-sha512}$  (P[0], salt)
    state  $\leftarrow \text{hmac-sha512}$  (salt_hash, pwd)
    for i  $\leftarrow 0$  to i < 3 do
        for j  $\leftarrow 0$  to j < sbbox_words, j+=SHA512_DIGEST_LENGTH do
            sbbox[i][j]  $\leftarrow \text{hmac-sha512}$  (salt_hash, state)
            state  $\leftarrow$  sbbox[i][j]
        end for
    end for
    key_hash  $\leftarrow \text{hmac-sha512}$  (state, pwd)
    expandkey (salt_hash, key_hash)
    count  $\leftarrow 2^{t\_cost}$ 
    do
        expandkey (null, salt_hash)
        expandkey (null, key_hash)
    while (--count)
    ctext  $\leftarrow$  "Drab as a fool, aloof as a bard."
    blockcnt  $\leftarrow \text{ceil}$  (outlen / SHA512_DIGEST_LENGTH)
    do
        for i  $\leftarrow 0$  to i < 64 do
            ecb_encrypt (ctext)
        end for
        output .=  $\text{hmac-sha512}$  (salt_hash, ctext)
    while (--blockcnt)
    return output
```

```
function encipher (L, R)
    for i  $\leftarrow 0$  to i < 18, i+=2 do
        L  $\oplus$  P[i]
        R  $\oplus$  f(L)
        R  $\oplus$  P[i+1]
        L  $\oplus$  f(R)
    end for
    L  $\oplus$  P[16]
    R  $\oplus$  P[17]
    swap (L, R)
```

```

function expandkey (data, key)
  for i  $\leftarrow$  0 to i < 18 do
    P[i]  $\leftarrow$  P[i]  $\oplus$  key[i]
  end for
  for i  $\leftarrow$  0 to i < 18, i+=2 do
    L  $\oplus$  data[i%keylength]
    R  $\oplus$  data[(i+1)%keylength]
    encipher (L, R)
    P[i]  $\leftarrow$  L, P[i+1]  $\leftarrow$  R
  end for
  for i  $\leftarrow$  0 to i < 3 do
    for j  $\leftarrow$  0 to j < sbox_words do
      L  $\oplus$  data[i%keylength]
      R  $\oplus$  data[(i+1)%keylength]
      encipher (L, R)
      sbox[i][j]  $\leftarrow$  L
      sbox[i][j+1]  $\leftarrow$  R
    end for
  end for

function f (x)
  log2_sbox_words  $\leftarrow$  2(m_cost + 5)
  return ((sbox[0][x >> (64-log2_sbox_words)]  $\oplus$ 
    sbox[1][x >> (48-log2_sbox_words) & (log2_sbox_words-1)]) +
    sbox[2][x >> (32-log2_sbox_words) & (log2_sbox_words-1)])  $\oplus$ 
    sbox[3][x >> (16-log2_sbox_words) & (log2_sbox_words-1)]);

function ecb_encrypt (data)
  for i  $\leftarrow$  0 to i < data_length, i+=16 do
    L  $\leftarrow$  data[i] .. data[i+7]
    R  $\leftarrow$  data[i+8] .. data[i+15]
    encipher (L, R)
    data[i] .. data[i+7]  $\leftarrow$  L
    data[i+8] .. data[i+15]  $\leftarrow$  R
  end for

```

Initial Security Analysis

Cryptographic security of Pufferfish is based on that of HMAC-SHA512 and Blowfish. While unlikely, the modifications made to Blowfish may render it non-cryptographic. However, a catastrophic failure of Pufferfish’s dynamic S-boxes or modified F function would not affect Pufferfish’s cryptographic properties so long as HMAC-SHA512 remains unbroken.

Pufferfish’s design also imposes no limits on salt size or password length, and has no issues handling inputs of various encodings.

Efficiency Analysis

Due to Pufferfish's use of 64-bit words, efficiency is improved 2x on 64-bit hardware compared to bcrypt at an m_cost of 2 (4 KiB) and the same t_cost , even with the additional SHA-512 overhead. Performance scales linearly compared to bcrypt depending on the m_cost . For example, at an m_cost of 4 (16 KiB), Pufferfish is only 2x slower than bcrypt at the same t_cost (4x slowdown from 4x the S-box size, 2x speedup from the move to 64-bit.)

Pufferfish inherits bcrypt's GPU resistance as it retains the small-but-frequent pseudo-random memory access patterns. And as 64-bit arithmetic performance is poor on current GPUs, the move to 64-bit increases GPU resistance compared to bcrypt. The additional SHA-512 operations add a small bit of GPU resistance as well, since SHA-512 performs poorly on GPU (only ~ 128x speedup on GPU compared to e.g. ~ 800x speedup on MD5.)

Intellectual Property Statement

Pufferfish has been placed in the public domain and is, and will remain, available worldwide on a royalty-free basis. The designer is unaware of any patent or patent application that covers the use or implementation of the submitted algorithm.

Hidden Weakness Statement

There are no deliberate or hidden weaknesses in Pufferfish.

References

[1] The Blowfish Encryption Algorithm, <https://www.schneier.com/blowfish.html>

[2] bcrypt Algorithm,
<https://www.usenix.org/legacy/publications/library/proceedings/usenix99/provos/provos.html/node5.html>

[3] Eksblowfish Algorithm,
<https://www.usenix.org/legacy/publications/library/proceedings/usenix99/provos/provos.html/node4.html>