Yarn password hashing function

Evgeny Kapun
abacabadabacaba@gmail.com

1 Introduction

I propose a password hashing function Yarn. This is a memory-hard function, however, a prime objective of this function is not to satisfy any theoretical definition of memory hardness, but rather to be as efficient to compute on a modern x86 processor as possible, compared with other hardware. Ideally, a modern x86 processor should be the most cost-effective device for computing this function. Therefore, it may be useful on a wide range of desktop, laptop and server machines.

This is achieved by two means. Firstly, this function makes heavy use of `aesenc` x86 instruction and instruction-level parallelism, which is intended to maximize utilization of execution units present in modern x86 processors. Secondly, this function repeatedly performs memory lookups, such that the address of each lookup depends on the result of the previous one. Therefore, access to some memory with sufficiently low latency is necessary to compute this function efficiently.

This function has a number of parameters which can be tweaked to achieve the best possible resource utilization.

2 Schematic description

The operation of Yarn consists of five phases. The first phase is an application of BLAKE2b hash function to the password. However, the entire final state is retained (Figure 1).

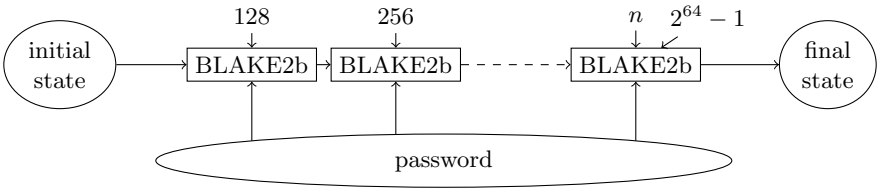

Figure 1: The first phase. Boxes labeled "BLAKE2b" correspond to calls to the BLAKE2b compression function, not the entire hash function.

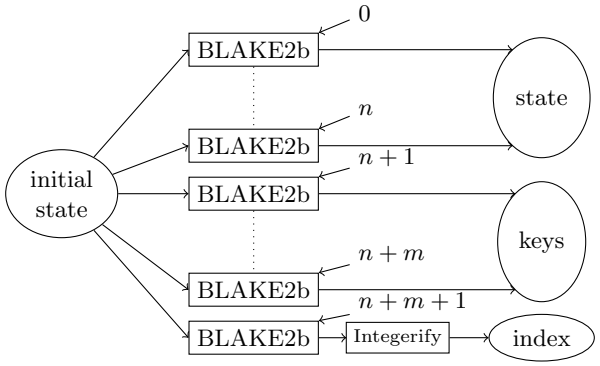

Figure 2: The second phase. Here, the initial state is the same as the final state of the first phase.

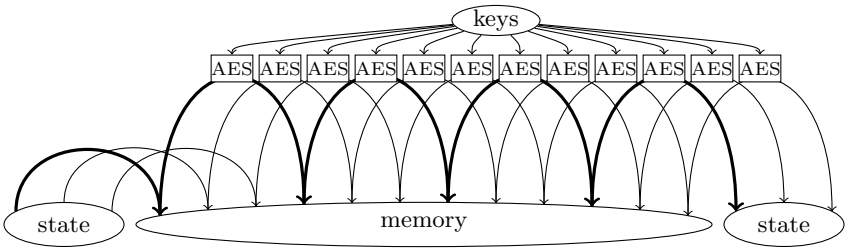

Figure 3: The third phase. The old value of the state is on the left, the new value is on the right. Each box labeled "AES" represents an instance of AES-like encryption, which consists of multiple rounds.

In the second phase, the resulting state is expanded to produce enough pseudorandom data for the subsequent phases. This is done by applying BLAKE2b compression function multiple times to the same state, but using a different tweak each time. For these calls, message block is set to all zeros. The result is split into three parts: the state, the key, and the index (Figure 2).

In the third phase, a large array is initialized with pseudo-random data. It will be subsequently queried at pseudorandom addresses to achieve memory-hardness. However, elements are initialized sequentially. During the initialization, an AES-like algorithm is used, which will be described later. The values "state" and "keys" from the previous phase are used, moreover, the "state" is also updated (Figure 3).

The majority of the computation time is expected to be spent in the fourth phase. In this phase, multiple evaluations of `aesenc` primitive are interleaved with random-address memory accesses. The evaluations are structured to permit limited parallelism to better utilize CPU resources. Memory accesses can also be done in parallel with `aesenc` computations (Figure 4).

In the fifth phase, the resulting state is hashed, starting from BLAKE2b

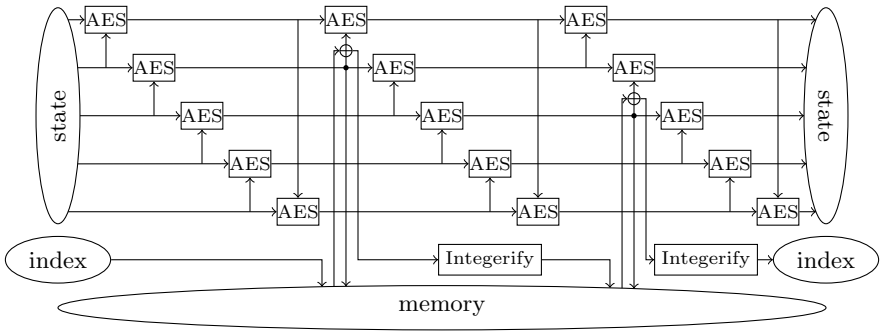

Figure 4: The fourth phase. Each box labeled "AES" represents an `aesenc` evaluation, where a left input is a data block and a top or a bottom input is a round key.

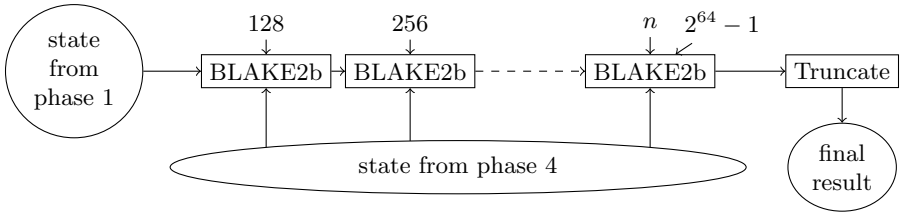

Figure 5: The fifth phase. Almost like BLAKE2b calculation, but the initial state is different.

final state produced in the first phase. The resulting state is truncated to the desired output length and returned. Therefore, the result is a slightly modified BLAKE2b hash of the concatenation of the password and the state produced in phase 4. The difference is that both the password and the state from phase 4 are padded, and that the counters and finalization flags are set like there were two independent hash calculations (Figure 5).

3 Specification

Notation: operation of various functions below is described using pseudocode. In pseudocode, all values are either arbitrary-precision integers or byte sequences (or arrays of them). Array indices are zero-based. Operators are represented using mathematical notation, with **and**, **xor**, **shl**, and **shr** being bitwise operators, and || representing string concatenation.

The function takes the following parameters:

- Password (*in*) – a string up to $2^{128} - 1$ bytes in length.
- Salt (*salt*) (optional) – a string up to 16 bytes in length.

- Personalization string (*pers*) (optional) – a string up to 16 bytes in length.
- Output length (*outlen*) – a positive integer not exceeding 64.
- Time cost (*t_cost*) – a nonnegative integer.
- Memory cost (*m_cost*) – a nonnegative integer not exceeding 124. Memory requirement is 2^{m_cost+4} bytes.
- Internal parallelism level (*par*) – a positive integer not exceeding $2^{124} - 1$.
- Number of rounds for the initial phase (*initrnd*) – a positive integer not exceeding $2^{124} - 1$.
- Number of iterations between memory accesses (*m_step*) – a positive integer.

Definitions:

Blake2b is a BLAKE2b hash function[1]. It consists of the following steps:

function Blake2b(*in*, *outlen*, *salt*, *pers*)
  $h \leftarrow$ Blake2b_GenerateInitialState(*outlen*, *salt*, *pers*)
  $h \leftarrow$ Blake2b_ConsumeInput(h, *in*)
  return Truncate(h, *outlen*)
end function

These steps will be used separately, as well as the BLAKE2b compression function: Blake2b_Compress(h, m, t_0, t_1, f_0, f_1).

This function will be used to produce arbitrary number of pseudorandom bytes from a BLAKE2b state:

function Blake2b_ExpandState(h, *outlen*)
  out $\leftarrow$ empty string
  for $i = 0, \ldots, \lceil \frac{outlen}{64} \rceil - 1$ **do**
    out $\leftarrow$ *out* || Blake2b_Compress(h, 128 null bytes,
      Low64Bits(i), High64Bits(i), 0, 8 `0xff` bytes)
  end for
  return Truncate(*out*, *outlen*)
end function

AESEnc is a round of AES encryption, equivalent to x86 `aesenc` instruction. It can be represented as:

function AESEnc(*data*, *key*)
  return AESPermutation(*data*) **xor** *key*
end function

Here, AESPermutation is a combination of SubBytes, ShiftRows and MixColumns AES steps.

The following function is similar to AES encryption, but simpler. *keys* is an array of *initrnd* AES round keys.

[1] `http://blake2.net/`

function AESPseudoEncrypt($data, keys$)
  for $i = 0, \ldots, initrnd - 1$ **do**
    $data \leftarrow$ AESEnc($data, keys[i]$)
  end for
  return $data$
end function

Unlike the real AES, there are no distinct initial and final rounds.

There are two main arrays used in Yarn function: *state* and *memory*. *state* consists of *par* 16-byte blocks, and *memory* consists of 2^{m_cost} 16-byte blocks. This function rotates *state* one block to the left:

function RotateState($state$)
  return $state[1 \ldots par - 1] \,||\, state[0]$
end function

The function Integerify turns a 16-byte block into a valid index in the *memory* array. It represents the entire block as a little-endian integer, then discards its 4 least significant bits and everything above its $4 + m\_cost$ least significant bits. This can be summarized as follows:

function Integerify($block$)
  $n \leftarrow$ AsLittleEndianInteger($block$)
  return $(n$ **shr** $4)$ **and** $((1$ **shl** $m\_cost) - 1)$
end function

The Yarn function consists of five phases. In the first phase, the password is hashed using BLAKE2b hash function. In the second phase, the final state is used to derive pseudorandom initial values for *state*, *keys* and *index*. During the third phase, the *memory* array is filled with *par* interleaved OFB-mode keystreams computed using the AESPseudoEncrypt function. During the fourth phase, multiple AESEnc computations and random memory accesses are performed such that they can only be parallelized to a specific extent. During the fifth phase, the result of the fourth phase is compressed with the hash state produced in the first phase to obtain the final value of the hash.

function Yarn($in, salt, pers, outlen, t\_cost, m\_cost, par, initrnd, m\_step$)
  // Phase 1 – BLAKE2b hashing
  $h \leftarrow$ Blake2b_GenerateInitialState($outlen, salt, pers$)
  $h \leftarrow$ Blake2b_ConsumeInput(h, in)
  // Phase 2 – generation of the initial state
  $expanded\_h \leftarrow$ As16ByteBlocks(
      Blake2b_ExpandState($h, 16 \cdot (par + initrnd + 1)$))
  $state \leftarrow expanded\_h[0 \ldots par - 1]$
  $keys \leftarrow expanded\_h[par \ldots par + initrnd - 1]$
  $index \leftarrow$ Integerify($expanded\_h[par + initrnd]$)
  // Phase 3 – memory initialization
  for $i = 0, \ldots, 2^{m\_cost} - 1$ **do**
    $memory[i] \leftarrow state[0]$
    $state[0] \leftarrow$ AESPseudoEncrypt($state[0], keys$)
    $state \leftarrow$ RotateState($state$)

```
    end for
    // Phase 4 – main phase
    for i = 0, ..., t_cost − 1 do
        block ← state[1 mod par]
        if i mod m_step = m_step − 1 then
            block2 ← memory[index]
            memory[index] ← block
            block ← block xor block2
            index ← Integerify(block)
        end if
        state[0] ← AESEnc(state[0], block)
        state ← RotateState(state)
    end for
    // Phase 5 – finalization
    h ← Blake2b_ConsumeInput(h, AsBytes(state))
    return Truncate(h, outlen)
end function
```

4 Security analysis

The security of Yarn function rests on the security of BLAKE2b function, as well as on properties of AES permutation. The hash is computed by using BLAKE2b compression function to compress password, salt, and the result of a memory-hard computation into a single value. The compression is structured such that Yarn has optimal preimage and collision resistance properties with respect to the password, assuming that the compression function is ideal. The proof is similar to that of the security of BLAKE2b function. I believe that Yarn also has the indifferentiability property of BLAKE2b.

Yarn function has certain collisions on auxiliary parameters. For example, appending zero bytes to salt or personalization string doesn't change the value. However, if the function is used as intended, these collisions should have no security impact.

The memory-hard part uses the final state of BLAKE2b hash of the password to derive the initial state. Because of this, it is not possible to perform this part of computation, which takes the majority of time, for multiple related password/salt pairs (such as those differing only in salt) more efficiently than for unrelated pairs. So, an attacker, who may want to compute hashes of the same password with multiple salts, won't be able to do such computation more efficiently than several independent hash computations.

During the third phase, the memory area is filled with pseudorandom content produced using AES algorithm. It is not necessary for it to be strongly random for security, because the memory is written to during the fourth phase, which prevents time-memory trade-offs. However, if the initial content were too easy to compute, this could be used to avoid storing it at least for the beginning of the fourth phase, when only few of the memory blocks have their value changed

from the initial. The algorithm is designed such that an attacker cannot quickly compute the initial value for a block at arbitrary index without precomputing and storing intermediate results, which reduces his advantage over a straightforward implementation.

5 Efficiency analysis

It is intended that the parameters are selected such that the computation time is dominated by the fourth phase. To this end, the value of *m_cost* should not be too small, and the value of *t_cost* should be at least several times larger than 2^{m_cost}. During this phase, `aesenc` instructions are repeatedly computed such that up to *par* AES permutations can be computed in parallel. This number should be chosen to match the number of `aesenc` instructions that the target CPU can compute in parallel. According to Intel optimization manual[1], modern Intel processors can evaluate approximately 8 AES instructions at a time on one core. Choosing too large value for *par* may benefit attackers using bitsliced AES implementation and should therefore be avoided.

In addition to AES computations, a number of memory lookups are performed using pseudorandom indices. These lookups are there to prevent an attacker without access to sufficient amounts fast memory from computing Yarn efficiently. Each index depends on the results of the previous lookup, so these lookups have to be performed sequentially. This makes the evaluation speed highly dependent on memory latency. At the same time, memory lookups can be performed in parallel with AES evaluations, making both AES units and memory fully utilized.

Yarn deliberately doesn't take advantage of multithreading. I think that in order to utilize multithreading, a higher level primitive should be used, which will compute multiple instances of a function like Yarn in parallel and combine the results.

Evaluating Yarn on GPU would not be very efficient, because GPUs are not optimized for AES evaluations. Also, if *m_cost* is large enough, the number of instances of Yarn that can be computed concurrently on a GPU would be limited by available memory, making GPU use even less efficient. To compute Yarn efficiently on FPGAs and ASICs, it would be necessary for them to have both fast memory controller and multiple AES units. However, a CPU has both already, so making a cost-effective FPGA or ASIC for Yarn would be tricky.

6 Disclaimers

This scheme and the accompanying code doesn't contain any deliberately introduced deficiencies or weaknesses.

[1] `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`