POMELO
A Password Hashing Algorithm
(Version 2)

Designer and Submitter: Hongjun Wu

Division of Mathematical Sciences
Nanyang Technological University
wuhongjun@gmail.com

(last modified on 2015.04.13)

Contents

Chapter 1

Specifications of POMELO

1.1 Operations, Variables and Functions

The operations, variables and functions used in POMELO are defined below.

1.1.1 Operations

The following operations are used in POMELO:

$+$	:	addition modulo 2^{64}
$\oplus$	:	bit-wise exclusive OR
$\&$	:	bit-wise AND
$\|$	:	concatenation
$<<$	:	left-shift
$>>$	:	right-shift
$<<<$	:	left-rotation. $x <<< n$ means that $(x << n) \oplus (x >> (64 - n))$, where x is a 64-bit integer, n is a non-negative integer not larger than 64.

Let X and Y be two 256-bit words. $X = x_3 \| x_2 \| x_1 \| x_0$, where each x_i is 64-bit. $Y = y_3 \| y_2 \| y_1 \| y_0$, where each y_i is 64-bit.

$+'$	:	$X +' Y = (x_3 + y_3) \| (x_2 + y_2) \| (x_1 + y_1) \| (x_0 + y_0)$.
$SHL256$	:	$SHL256(X, n) = (x_3 << n) \| (x_2 << n) \| (x_1 << n) \| (x_0 << n)$.
$ROTL256$	:	$ROTL256(X, n) = (x_3 <<< n) \| (x_2 <<< n) \| (x_1 <<< n) \| (x_0 <<< n)$.
$ROTL256\_64$	:	$ROTL256\_64(X) = x_2 \| x_1 \| x_0 \| x_3$.

Note that in C programming, if AVX2 instructions are available, the operations on 256-bit words can be implemented using the compiler intrinsics:

1. $X +' Y$ is implemented as _mm256_add_epi64(X, Y);
2. $X \oplus Y$ is implemented as _mm256_xor_si256(X, Y);
3. $SHL256(X, n)$ is implemented as _mm256_slli_epi64(X, n);
4. $ROTL256(X, n)$ is implemented as _mm256_xor_si256(_mm256_slli_epi64(x,n), _mm256_srli_epi64(x,(64-n)))
5. $ROTL256\_64(x)$ is implemented as _mm256_permute4x64_epi64((x), _MM_SHUFFLE(2,1,0,3))

1.1.2 Variables

m_cost	:	the parameter used to adjust the memory size. $0 \leq m\_cost \leq 25$
pwd	:	the password
pwd_size	:	the password size in bytes. $0 \leq t \leq 256$.
$random\_number$	:	a 64-bit pseudo random number.
S	:	the state. The state size is 2^{13+m_cost} bytes.
$S8[i]$	:	the ith byte of the state.
$S64[i]$	:	the ith 64-bit word of the state. $S64[i] = S8[8i+7] \parallel S8[8i+6] \parallel S8[8i+5] \parallel S8[8i+4] \parallel S8[8i+3] \parallel S8[8i+2] \parallel S8[8i+1] \parallel S8[8i]$
$S[i]$	:	the ith 256-bit word of the state. $S[i] = S64[4i+3] \parallel S64[4i+2] \parallel S64[4i+1] \parallel S64[4i]$
$state\_size$	:	the state size in bytes. $state\_size = 2^{13+m\_cost}$.
$salt$	:	the salt.
$salt\_size$	:	the byte size of salt. $0 \leq salt\_size \leq 64$.
t_cost	:	the parameter used to adjust the timing. $0 \leq t\_cost \leq 25$.
t	:	the output size in bytes. $1 \leq t \leq 256$.

1.1.3 Functions

Four state update functions are used in POMELO. Their specifications are given below. Note that $S[i]$ is the ith 256-bit element of the state, and there are $state\_size/32$ elements in the state.

State update function $\mathbf{F}(S, i)$:

$i0 = (i - 0) \bmod (state\_size/32)$;
$i1 = (i - 2) \bmod (state\_size/32)$;
$i2 = (i - 3) \bmod (state\_size/32)$;
$i3 = (i - 7) \bmod (state\_size/32)$;
$i4 = (i - 13) \bmod (state\_size/32)$;
$S[i0] = S[i0] +' (((S[i1] \oplus S[i2]) +' S[i3]) \oplus S[i4])$;
$S[i0] = ROTL256\_64(S[i0])$;
$S[i0] = ROTL256(S[i0], 17)$;

State update function G($S, i, random\_number$).

In this function, element $S[i]$ is updated; two table lookups are used to update two elements $S[index\_global]$ and $S[index\_local]$. At each step, the value of $index\_local$ is updated according to $random\_number$. In the i-th step, the value of $index\_global$ is updated according to $random\_number$ if i is a multiple of 32; otherwise, the value of $index\_global$ is incremented by 1.

The range of $index\_local$ is $[i-4096, i+4096)$, and is within the range of the state; the range of $index\_global$ is the whole state. Note that if the state size is not more than 2^{18} bytes (m_cost is not larger than 5, i.e., there are at most 8192 256-bit elements in the state), the range of $index\_local$ and $index\_global$ are the same.

G($S, i, random\_number$)
{
    F(S, i);

    //update index_local and index_global
    $index\_local = (i-4096+(random\_number \bmod 8192)) \bmod (state\_size/32)$;
    if ($i \bmod 32 == 0$)
        $index\_global = (random\_number \gg 16) \bmod (state\_size/32)$;
    endif;
    $index\_global = (index\_global + 1) \bmod (state\_size/32)$;

    //table lookup S[index_local], here i0 is (i mod statesize/32)
    $S[i0] = S[i0] +' SHL256(S[index\_local], 1)$;
    $S[index\_local] = S[index\_local] +' SHL256(S[i0], 2)$;

    //table lookup S[index_global]
    $S[i0] = S[i0] +' SHL256(S[index\_global], 1)$;
    $S[index\_global] = S[index\_global] +' SHL256(S[i0], 3)$;

    //update random_number
    $random\_number+ = (random\_number \ll 2)$;
    $random\_number = (random\_number \lll 19) \oplus 3141592653589793238ULL$;
}

Note that in function G($S, i, random\_number$), $index\_local$ and $index\_global$ are updated independent of the input password, so the table lookups in this function are not affected by the cache-timing side-channel attack.

State update function H$(S, i, random_number)$:

Function H is very similar to function G. The only difference is that at the end of function H, *random_number* is updated according to the secret state. So the table lookups in function H are affected by the cache-timing side-channel attack.

H$(S, i, random_number)$
{
    F(S, i);

    //update index_local and index_global
    $index\_local = (i{-}4096{+}(random\_number \bmod 8192)) \bmod (state\_size/32)$;
    if ($i \bmod 32 == 0$)
        $index\_global = (random\_number \gg 16) \bmod (state\_size/32)$;
    endif;
    $index\_global = (index\_global + 1) \bmod (state\_size/32)$;

    //table lookup S[index_local], here i0 is (i mod statesize/32)
    $S[i0] = S[i0] +' SHL256(S[index\_local], 1)$;
    $S[index\_local] = S[index\_local] +' SHL256(S[i0], 2)$;

    //table lookup S[index_global]
    $S[i0] = S[i0] +' SHL256(S[index\_global], 1)$;
    $S[index\_global] = S[index\_global] +' SHL256(S[i0], 3)$;

    //update random_number as a 64-bit word of the state.
    //here i3 = (i-7) mod state_size/32
    //S64[i] indicates the ith 64-bit element of the state.
    $random\_number = S64[4 * i3]$;
}

1.2 Hashing the password and salt

The hashing algorithm is given below. Note that $S[i]$ is the ith 256-bit element, $S8[i]$ is the ith byte of the state. There are $2^{8+m\_cost+t\_cost}$ 256-bit elements in the state.

1. Initialize the state S to 0, the state_size is 2^{13+m_cost} bytes.

2. //load the password, salt, and the input/output sizes into the state.
   Let $S8[i] = pwd_i$ for $i = 0$ to $pwd\_size - 1$;
   Let $S8[pwd\_size + i] = salt[i]$ for $i = 0$ to $salt\_size - 1$;
   Let $S8[384] = pwd\_size$ mod 256;
   Let $S8[385] = pwd\_size/256$;
   Let $S8[386] = salt\_size$;
   Let $S8[387] = output\_size$ mod 256;
   Let $S8[388] = output\_size/256$;

   //introducing random constants to the state using Fibonacci sequence.
   Let $S8[392] = 1$;
   Let $S8[393] = 1$;
   Let $S8[i] = (S8[i-1] + S8[i-2])$ mod 256 for $i = 394$ to 415;

3. // expand the data into the whole state.
   for $i = 13$ to $2^{8+m\_cost} - 1$, do: $\mathrm{F}(S, i)$;

4. // update the state using function G
   // (involving password-INdependent random memory accesses)
   $random\_number = 123456789ULL$;
   for $i = 0$ to $2^{7+m\_cost+t\_cost} - 1$, do: $\mathrm{G}(S, i, random\_number)$;

5. // update the state using function H
   // (involving password-dependent random memory accesses)
   for $i = 2^{7+m\_cost+t\_cost}$ to $2^{8+m\_cost+t\_cost} - 1$, do: $\mathrm{H}(S, i, random\_number)$;

6. // update the state using F
   for $i = 0$ to $2^{8+m\_cost} - 1$, do: $\mathrm{F}(S, i)$;

7. The hash output is given as the last t bytes of the state S ($t <= 256$):
   $S8[state\_size - t] \parallel S8[state\_size - t + 1] \parallel \cdots \parallel S8[state\_size - 1]$.

1.3 Recommended Parameters

We recommend $5 \leq m\_cost + t\_cost \leq 25$. The memory size of POMELO is 2^{13+m_cost} bytes, i.e., 2^{8+m_cost} 256-bit words. There are $2^{7+m\_cost+t\_cost}$ function G and $2^{7+m\_cost+t\_cost}$ function H.

The recommended memory size ranges from 8KB ($m\_cost = 0$) to 256GB ($m\_cost = 25$). When $m\_cost + t\_cost = 5$, it is very fast to compute the password hashing. When $m\_cost + t\_cost = 25$, we get very high security, but it is also very expensive to compute the password hashing.

Choosing the proper values of m_cost and t_cost depends on the requirements of applications. A user can find more information in Section 3.1 on how to choose the proper parameters (or to test the POLEMO code by adjusting the value of m_cost and t_cost).

If a user wants to compute the hash in a fast way with large memory, the user may use $m\_cost = 15$, $t\_cost = 0$ (2^{28} bytes of memory).

1.4 Efficient implementation

The 'if' selection statement in function G and H can be removed when we implement 32 G functions (or H functions) in a 'for' loop.

The modular operation in POMELO can be implemented using the bit-wise AND operation since the divisors are the power of 2.

1.5 POMELO as key derivation function

POMELO can be used as a key derivation function (KDF) to derive a key from a password. There are several ways to generate a key with arbitrary length from POMELO. The algorithm described in this section uses the POMELO algorithm in key derivation with slight modification to POMELO. When POMELO is used for KDF, the value of $S8[389]$ is set to 1 in the Step 2 of POMELO. We call this modified POMELO as POMELO'. The reason that we set $S8[389]$ to 1 is to prevent the related-cipher attack [12] so that even if a password is used in both password hashing algorithm and key derivation algorithm (and the salts are the same), the password security would not be significantly affected; otherwise, the password image of password hashing may be used to compromise the security of the derived key. The API of POMELO can be slightly modified to include this variant for KDF.

Suppose that the key K consists of n bytes, $K = k_0 \parallel k_1 \parallel \cdots \parallel k_{n-2} \parallel k_{n-1}$. There is no restriction on the value of n. But we expect that in all the applications, n is much less than 2^{64}. POMELO-KDF is given below. If a password is used in POMELO-KDF to generate multiple keys (instead of a long key), the user should generate a different salt for each invocation of POMELO-KDF.

//POMELO-KDF: Derive an n-byte key K from password;
//POMELO' is iterated to generate a long key;
//Each invocation of POMELO' always generates 256 bytes output,
//half of the outputs are key bytes.

1. Modify the POMELO algorithm so that S8[389] is set to 1 in Step 2 of POMELO. The modified POMELO is called POMELO'.
2. for $i = 0$ to $\lceil \frac{n}{128} \rceil - 1$
   {
   output_size = 256;
   POMELO'(output,output_size,pwd,pwd_size,salt,salt_size,m_cost,t_cost);
   pwd = output; //use the original password only once.
   pwd_size = 256;
   $k_{128i} \parallel k_{128i+1} \parallel \cdots \parallel k_{128i+127}$ are the second half of output;
   }
3. The key K is the first n key bytes: $k_0 \parallel k_1 \parallel \cdots \parallel k_{n-2} \parallel k_{n-1}$.

The parameters of *m_cost* and *t_cost* in POMELO-KDF are identical to that in POMELO. In POMELO-KDF, the key length n is not an input, so the key can be derived without knowing the total key size. The password is used only once in POMELO-KDF so that the program does not need to keep the password in memory during the key derivation process.

Chapter 2

Security Analysis

We have the following security claims based on our initial security analysis:

Claim 1. It is impossible to recover the password from the hash output faster than trying those possible passwords, i.e., POMELO is one-way.

Claim 2. POMELO is strong against the attacks that intend to bypass the large memory in POMELO.

Claim 3. POMELO is strong against the attacks using GPU and dedicated hardware. It is due to the use of large memory in POMELO, and it is difficult to attack using smaller memory space.

Claim 4. POMELO is strong against the cache-timing attack [4] since the first half of POMELO uses password independent memory accesses.

2.1 Randomness, Preimage/Collision Resistance

For any password hashing algorithm, the randomness preimage/collision resistance can be achieved easily. The reason is that in general we can perform a lot of computations and use a large state in the algorithm, and the password is secret to the attackers. Thus designing password hashing algorithm is much easier than designing cryptographic hash function when we are talking about randomness, preimage/collision security.

Note that we used a Fibonacci sequence to initialize 24 bytes of the state. The Fibonacci sequence is used to prevent the symmetry structure of POMELO being exploited in an attack.

2.1.1 Differential cryptanalysis

Differential cryptanalysis [7] is powerful against symmetric ciphers. To design a password hashing algorithm, we should first ensure that the input difference

cannot be eliminated easily in the state; otherwise, the collision in the state may be exploited to bypass most of the operations in a cipher, such as the differential attacks against stream ciphers Py [13] and ZUC [14]. In POMELO, the password and salt are expanded into a state with size at least 8192 bytes through an **invertible** non-linear feedback register (Step 3 in Sect. 1.2). Step 4 is also invertible. So there is no collision in the state in Step 3 and 4 of POMELO.

Step 3, 4, 5, and 6 of POMELO contribute to the differential propagation in POMELO. The addition operations in Step 3 and 6 affect the differential propagation; the table lookups and additions in Step 4 and 5 affect the differential propagation.

We first look at Step 3. Suppose that there is an input difference. We analyze the simple feedback function of POMELO to identify the most efficient differential path. Our simulation shows that the most efficient differential path occurs when 13 consecutive elements are all zero except that one of most significant bits is one. There are 92 difference bits passing through addition in the nonlinear feedback function in every 100 feedback steps; there are 212 difference bits passing through additions in every 150 feedback steps; and there are 412 difference bits passing through additions in every 200 feedback steps; and there are 587 difference bits passing through the additions in every 250 feedback steps. (When we are counting the difference bits in addition, we ignore the difference bits at the most significant bit positions; and we simply ignore the difference bits if they occur at the same positions of two addends although they also contribute to the differential cryptanalysis). Each difference bit introduces differential probability 2^{-1}, so we claim that each feedback step introduces differential probability 2^{-1} on average. For $m\_cost = 0$, we have the smallest number of feedback steps (which is 243), so the differential probability of Step 3 is at most 2^{-243}.

We proceed to consider Step 4. The password hashing algorithm cannot be too fast, so we have the restriction that $m\_cost + t\_cost >= 5$ for POMELO. Step 4 thus involves at least 4096 feedback steps. Each feedback step in Step 4 is much stronger than that in Step 3 since two table lookups are used in each feedback step of Step 4. The difference bit(s) from "random" location in the state would be introduced into each feedback step of Step 4, and much more difference bits would be introduced. Thus we can claim that the differential probability of Step 4 is at most 2^{-4096}.

Now consider Step 5. Each feedback step in Step 5 is stronger than that in Step 4 since password dependent table lookups are used in each feedback step of Step 5. Thus we can claim that the differential probability of Step 5 is at most 2^{-4096}.

Each feedback step in Step 6 is stronger than that in Step 3 since two additions are involved in a feedback step of Step 6; while effectively there is only one addition in a feedback step of Step 3.

Combining the above analysis, the differential probability of POMELO is less than 2^{-8000}. It shows that POMELO is very strong against the differential attack.

2.1.2 Linear cryptanalysis

To simplify the linear cryptanalysis [9], we consider only the linear approximation in Step 5 in which we lookup a secret and changing Sbox (part of the state). To simplify the analysis further, we consider only the local table lookups. For $m\_cost = 8$, the Sbox is the smallest 8bit-to-256bit Sbox.

In the expansion stage, 5 of those 14 elements are linearly related through the feedback function. In the following, we consider the linear relation in 14 consecutive elements since the output size is at most 256 bytes (8 elements). In every feedback step in Step 5, each element is added with an output of the Sbox. After 14 feedback steps, the new additional bias of the linear relation of those 5 elements becomes $2^{-8\times4} = 2^{-32}$ (since those 5 table lookups may provide special outputs satisfying the linear relation; however, once an output is chosen, the positions of other four outputs are fixed, so the probability is 2^{-32}). In the next 14 feedback steps, the new additional bias of the linear relations again becomes 2^{-32}. Since Step 5 consists of at least 4096 feedback steps, the new additional bias of the linear relation of Steps 5 becomes less than $2^{4096/14-1} \times (2^{-32})^{4096/14} = 2^{-9071}$ according to the pilling-up lemma. The linear bias is sufficiently small, and POMELO is very strong against the linear cryptanalysis.

2.1.3 Slide attack

Slide attack [8] is powerful for bypassing a lot of operations if the cipher consists of many identical steps/rounds. A typical approach to defend against slide attack is to introduce different constants at each step/round. Slide attack is not a threat to POMELO since we use table lookups to access the state: any slide of the elements in the state would result in completely different table lookup outputs.

2.1.4 Randomness

Since POMELO is strong against the differential, linear cryptanalysis and slide attack, the output of POMELO is random.

2.1.5 Preimage/Collision Resistance

Since POMELO is strong against the differential, linear cryptanalysis, the output of POMELO is random.

2.2 Low Memory Attack

An important design goal of password hashing algorithm is to prevent the large memory state being bypassed easily in password cracking. Writing to random

memory locations is an effective way to resist the low-memory attack. A password hashing scheme that is designed to resist the low-memory is the scrypt [10].

Our analysis shows that if an attacker uses only half of the memory in the attack against POMELO (for the smallest value of $t\_cost = 0$), the number of operations would be increased by at least 128 times; if only one quarter of the memory is used in the attack, the number of operations would be increased by at least 16384 times. The details of the attack are given in this section.

To make the analysis simple, we assume that accessing any data from Step 3 (data expansion) requires negligible amount of storage and computation. The following three steps (for $t\_cost = 0$) should be computed in the attack.

4. $random\_number = 123456789ULL$;
   for $i = 0$ to $2^{7+m\_cost} - 1$, do: $\mathrm{G}(S, i, random\_number)$;
5. for $i = 2^{7+m\_cost}$ to $2^{8+m\_cost} - 1$, do: $\mathrm{H}(S, i, random\_number)$;
6. for $i = 0$ to $2^{8+m\_cost} - 1$, do: $\mathrm{F}(S, i)$;

2.2.1 Use half of the memory in the attack

Note that in Step 4 and 5, on average each state element is updated once through *index_global*; and each state element is updated once through *index_local*. Suppose that the memory being used in the attack is only half of the state, we consider that at the $16384n$-th step of Step 4 and 5, the attacker stores the elements $S[j]$ for $16384n - 4096 < j < 16384n + 4096$ (the data size can be slightly less than 256KB). From these partial states, the attacker may have the chance to construct the rest of the state.

In Step 6, we need the whole state that has been updated in Step 4 and 5. Note that many 1KB data segments (of 32 steps) are updated in function G or H through the global table lookup (note that the whole state is updated once in this way on average). In Step 6, such an updated 1KB data segment should be computed from a remote 256KB partial state S^x_{256KB} using G or H (we assume that the location of that partial state was stored with negligible amount of memory). To generate such a 256 KB partial state from some stored partial state ($S[j]$ with $16384n - 4096 < j < 16384n + 4096$), on average 8192 steps are needed (since the distance between two stored partial states is 16384 steps, and that only forward computation from a partial state is cost effective). Thus the attack requires $8192/32 = 256$ times more steps. If we assume that the attacker can manage to save half of those updated 1KB data segments, the number of operations in the attack still increase by 128 times.

2.2.2 Use one quarter of the memory in the attack

Suppose that the memory being used in the attack is only one quarter of the state, we consider that at the $32768n$-th step of Step 4 and 5, the attacker stores the elements $S[j]$ for $32768n - 4096 < j < 32768n + 4096$ (the data size can be

slightly less than 256KB). From these partial states, the attacker may have the chance to construct the rest of the state.

In Step 6, we need the whole state that has been updated in Step 4 and 5. Note that many 1KB data segments (of 32 steps) are updated in function G or H through the global table lookup (note that the whole state is updated once in this way on average). In Step 6, such an updated 1KB data segment should be computed from a remote 256KB partial state S^x_{256KB} using G or H (we assume that the location of that partial state was stored with negligible amount of memory). To generate such a 256 KB partial state from some stored partial state ($S[j]$ with $32768n - 4096 < j < 32768n + 4096$), on average 16384 steps are needed (since the distance between two stored partial states is 32768 steps, and that only forward computation from a partial state is cost effective). Thus the attack requires $16384/32 = 512$ times more steps.

The actual complexity of the above attack is much higher since when we are computing those 16384 steps to reach a partial state S^x_{256KB} (from the stored partial state), 16384 global table lookups are again needed, and these table lookups can be very expensive. Suppose that 1KB data segment (for 32 steps) in the global table lookup is not the original data from Step 3, but it gets modified in Step 4 and 5 already (the chance is half), then we need to generate this 1KB data segment from some partial state ($S[j]$ with $32768n - 4096 < j < 32768n + 4096$), and it would cost another 16384 steps on average. If we assume that the attacker can manage to save half of those updated 1KB data segments (being modified in Step 4 and 5 already), the number of operations can be reduced by half. Thus the overall attack requires 512*(16384/2/32/2) = 65536 times more computation.

The above analysis shows that POMELO is very strong against the low memory attack.

2.3 SIMD Attack

The SIMD is efficient since the instruction decoding and control circuits can be greatly saved. In POMELO, function H makes the attack using SIMD expensive since the function H uses password-dependent memory accesses.

If an attacker wants to launch the SIMD type attack against POMELO, the attacker must first launch cache-timing attack to recover the state information leaked from function H, then use SIMD type attack against the first half of POMELO. It makes the attack much harder since likely it requires some malicious program running on the same machine as the POMELO algorithm.

Furthermore, POMELO allows the use of large memory in password hashing. The cost-saving of using SIMD becomes much less effective when compared to the memory cost (for example, 256 MB memory). So we believe that POMELO provides strong security against SIMD attack.

Chapter 3

Efficiency Analysis

3.1 Software Performance

We implemented POMELO in C code. We tested the speed on Intel Core i7-4770K 3.5GHz processor (Turbo Boost 3.9GHz is enabled, 256KB Level 2 cache for each core, 8MB shared Level 3 memory cache, Memory Types DDR3-1333/1600) running 64-bit Ubuntu 14.04. The DRAM size is 16 Gigabytes. The compiler being used is gcc 4.8.2, and the optimization option "gcc -mavx2 -O3" is used. The code being tested uses AVX2 instructions (it is submitted together with this report).

The performance data are given in Table 3.1. POMELO is efficient even when the state size is very large. For example, when the state size of POMELO is one Gigabytes ($m\_cost = 17$, $t\_cost = 0$), it takes 1.1 seconds to hash a password. When the state size of POMELO is 256 Megabytes, it takes only 0.28 seconds ($m\_cost = 15$, $t\_cost = 0$) to hash a password.

We also implemented POMELO without using AVX2 or SSE instructions. The C code is submitted together with this report. This code is expected to run efficiently on most of the computing platform. For example, when the state size of POMELO is one Gigabytes ($m\_cost = 17$, $t\_cost = 0$), it takes 1.74 seconds to hash a password on the processor Intel Core i7-4770K.

Note that when the state size is large, the malloc takes significant amount of time. A server can reduce this cost by using one malloc for hashing multiple passwords. For example, when the state size is one Gigabytes, if 20 passwords are hashed with one malloc, it takes 0.92 seconds to hash a password on average (about 20% improvement in speed).

3.2 Performance on GPU and Hardware

In Pomelo, large state size can be used. The use of random memory accesses (read and write) makes it ineffective to bypass the memory restriction (i.e., it is not cost-effective to use memory smaller than the state to launch the pass-

word cracking attack). It is thus expensive to implement the password cracking against POMELO on GPU and dedicated hardware.

The functions H accesses (read and write) memory in a random way. It is thus expensive to utilize the SIMD power of GPU to crack passwords protected by POMELO.

However, in case that there is cache timing attack against POMELO, the attacker can retrieve partial information of the state, and use this partial information to bypass the second half of POMELO. But even in the presence of successful cache timing attack, the attacker still has to attack the first half of the algorithm. The first half of the POMELO is not vulnerable to the cache-timing attack, and it uses large amount of random memory accesses (function G), so it is difficult to develop efficient attacks on GPU since large memory is still needed.

3.3 Client-independent update

We believe that for any tunable password hashing algorithm, it is straightforward to achieve client-independent update. It is as simple as follows: we use the algorithm with the new parameters to hash the old password image generated with the old parameters. After the update, for any input password, the algorithm is applied twice: one with the old parameters; another with the new parameters.

Another approach to design the client-independent update is to simply increase the parameter (such as changing the value of t_cost parameter). POMELO does not use this approach. The reason is that POMELO should have strong resistance against the cache-timing attack, so we avoid mixing the password-independent table lookups and password-dependent table lookups (the first half of POMELO is strong against the cache timing attack). Increasing the value of t_cost in POMELO also increases the number of password-independent table lookups that must be involved once the cache-timing attack is successful. If we use the second approach to design client-independent update, POMELO's strength against the cache-timing attack would get affected.

Table 3.1: The timing of POMELO on Intel Core i7-4770K

m_cost	t_cost	state size (bytes)	timing (seconds)
2	0	2^{15}	0.00001
3	0	2^{16}	0.00002
4	0	2^{17}	0.00004
5	0	2^{18}	0.00008
6	0	2^{19}	0.00018
7	0	2^{20}	0.00036
8	0	2^{21}	0.00079
9	0	2^{22}	0.00158
10	0	2^{23}	0.00445
11	0	2^{24}	0.01242
12	0	2^{25}	0.034
13	0	2^{26}	0.069
14	0	2^{27}	0.140
15	0	2^{28}	0.281
16	0	2^{29}	0.565
17	0	2^{30}	1.136
18	0	2^{31}	2.277
19	0	2^{32}	4.563
20	0	2^{33}	9.137
2	18	2^{15}	1.197
3	17	2^{16}	1.225
4	16	2^{17}	1.242
5	15	2^{18}	1.322
6	14	2^{19}	1.442
7	13	2^{20}	1.490
8	12	2^{21}	1.522
9	11	2^{22}	1.520
10	10	2^{23}	2.657
11	9	2^{24}	4.335
12	8	2^{25}	4.780
13	7	2^{26}	5.012
14	6	2^{27}	5.250
15	5	2^{28}	5.450
16	4	2^{29}	5.655
17	3	2^{30}	5.937
18	2	2^{31}	6.425
19	1	2^{32}	7.340
20	0	2^{33}	9.137

Chapter 4

Features

- Simple design. POMELO is based on a simple non-linear feedback function F. Function G and H involves simple random memory accesses. POMELO does not rely on any existing hash function or cipher in the design.
- Easy to implement. The algorithm description can be easily translated into programming code.
- Easy to configure. The memory size is 2^{13+m_cost} bytes; the number of operations is proportional to $2^{m\_cost+t\_cost}$.
- Efficient. In POMELO, large memory can be used. For example, on the Intel i7-4770K processor, it takes 0.28 seconds to hash a password when 256 MB state is used.
- Strong against the low-memory attack. When $t\_cost = 0$, if only half of the memory is used in the attack, the computational cost is increased by at least 128 times; if only one quarter of the memory is used in the attack, the computational cost is increased by at least 65536 times.
- Strong against cache timing attack since the first half of POMELO uses password-independent memory accesses.
- Strong against attacks using GPU since large memory can be used, and password-dependent random memory accesses are used in the second half of POMELO.
- Frequent accesses to the 256KB CPU cache. In the attack using dedicated ASIC or FPGA, there is additional cost of implementing fast cache; otherwise, the attack speed would become slower for large state.
- Support large input/output sizes. The password is up to 256 bytes, the salt is up to 64 bytes, the output is up to 256 bytes.

- 66 extra input bytes are reserved for the extension of the algorithm (such as the inclusion of secret key). These 66 bytes can be assigned to $S8[320] \cdots S8[383]$, and $S8[390]$, $S8[391]$ in Step 2. (Note that byte $S8[389]$ is used for KDF.)

Chapter 5

Tweaks and Rationale

In the tweak, we process four 64-bit words in parallel, and we perform two memory accesses in each step of function G and H.

- Four 64-bit words are now processed in parallel.
  Reason: To efficiently use the 256-bit AVX2 instructions on the new generation CPUs. The SIMD instructions are used in a number of recent cryptographic designs, such as stream ciphers Salsa [5], Chacha [6], hash functions BLAKE [1], BLAKE2 [2], JH [15] and authenticated ciphers NORX [3] and MORUS [16].

- Two table lookups are used in each step of function G and H. (In the previous version of G and H, one table loop is used in every four steps of function G and H.)
  One table lookup is within the whole range of the state using *index_global* which is set to a pseudorandom number every 32 steps, then its value is increased by one every step.
  Another table lookup is within (at most) 256KB around the element $S[i]$ in the ith step. *index_local* is used, and its value is set to a pseudorandom number in every step.

  Reasons:
  1) Ensure that in every step, there are table lookups so that it is expensive for the attacks using SIMD.
  2) To speed up the memory access by using the DRAM row buffer. DRAM uses an 8KB row buffer (for every access to the DRAM, 8KB in that row gets read into the row buffer, and it is fast to access the data in the row buffer). For POMELO with large state size, using *index_global* in table lookup allows us to access data in the row buffer efficiently (1KB data at sequential addresses are accessed).
  3) Sequential data access is not good for resisting the attack using low memory, so we need to use *index_local* in table lookups to make the low memory attack expensive (the reason is to make it expensive to store the

partial state. The details are provided in the security analysis in Section 2.2). Although in every step *index_local* is updated to a pseudo random number, it is not expensive to loop up table using *index_local* since the CPUs have a relatively fast and large Level 2 cache.

- The nonlinear feedback function in function F is modified.
  Reason: It is because that we now changed the word size from 64-bit to 256-bit.

- The random number generation is changed.
  Reason: Simply allow the easy derivation of pseudorandom numbers (pseudorandom numbers are being used in table lookups).

Chapter 6

No hidden weakness

The designers of POMELO state here that there are no deliberately hidden weaknesses in POMELO.

Chapter 7

Intellectual property

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

Bibliography

[1] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan. BLAKE. NIST SHA-3 competition finalist, 2011.

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, Christian Winnerlein. BLAK2. https://blake2.net/

[3] Jean-Philippe Aumasson, Philipp Jovanovic, Samuel Neves. NORX. Submission to the CAESAR Competition, 2014.

[4] Dan J. Bernstein. Cache-timing attacks on AES. Technical Report, University of Illinois, 2005.

[5] Dan J. Bernstein. The Salsa20 family of stream ciphers. New Stream Cipher Designs, pages 84-97, 2008.

[6] Dan J. Bernstein. ChaCha, a variant of Salsa20. http://cr.yp.to/chacha.html

[7] Eli Biham, Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. Advances in Cryptology – CRYPTO'90, pp. 2 – 21.

[8] Alex Biryukov, David Wagner. Slide Attacks. Fast Software Encryption – FSE'99, pp.245 - 259.

[9] Mitsuru Matsui. "Linear cryptanalysis method for DES cipher". Advances in Cryptology - EUROCRYPT 1993.

[10] Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. BSE-Can'09, May 2009.

[11] Niels Provos and David Mazieres. A future-adaptable password scheme. In USENIX Annual Technical Conference, FRENIX Track, UNENIX, 1999.

[12] Hongjun Wu. Related-Cipher Attacks. Information and Communications Security – ICICS 2002, pp. 447 – 455.

[13] Hongjun Wu, Bart Preneel. Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy. Advances in Cryptology – EUROCRYPT 2007, pp. 276 – 290.

[14] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, San Ling. Differential Attacks against Stream Cipher ZUC. Advances in Cryptology – ASIACRYPT 2012, pp. 262 – 277.

[15] Hongjun Wu. The hash function JH. NIST SHA-3 competition finalist, 2011.

[16] Hongjun Wu and Tao Huang. MORUS – A Fast Authenticated Cipher. Submission to the CAESAR competition, 2014.