POMELO: A Password Hashing Algorithm

Designer and Submitter: Hongjun Wu

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

2014.03.31

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

$+$	:	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.

1.1.2 Variables

m_cost	:	the parameter used to adjust the memory size. $0 \leq m\_cost \leq 18$
pwd	:	the password
pwd_size	:	the password size in bytes. $0 \leq t \leq 128$.
S	:	the state. The state size is 2^{13+m_cost} bytes.
S_i	:	the ith byte of the state.
$S[i]$	:	the ith 64-bit word of the state. $S[i] = S_{8i} \parallel S_{8i+1} \parallel S_{8i+2} \parallel S_{8i+3} \parallel S_{8i+4} \parallel S_{8i+5} \parallel S_{8i+6} \parallel S_{8i+7}$
state_size	:	the state size in bytes. $state\_size = 2^{13+m\_cost}$.
salt	:	the salt.
salt_size	:	the byte size of salt. $16 \leq salt\_size \leq 32$.
t_cost	:	the parameter used to adjust the timing. $0 \leq t\_cost \leq 20$.
t	:	the output size in bytes. $32 \leq t \leq 128$.

1.1.3 Functions

Three state update functions are used in POMELO. Their specifications are given below. We repeat here that $S[i]$ is the ith 64-bit word of the state.

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

```
$i1 = (i - 1) \bmod (state\_size/8);$
$i2 = (i - 3) \bmod (state\_size/8);$
$i3 = (i - 17) \bmod (state\_size/8);$
$i4 = (i - 41) \bmod (state\_size/8);$
$S[i] = S[i] + (((S[i1] \oplus S[i2]) + S[i3]) \oplus S[i4]);$
$S[i] = S[i] <<< 17;$
```

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

```
if ($i \bmod 4 == 3$)
    $temp = 5^{j \times (state\_size/8)+i};$
    $index = (temp + (temp >> 32)) \bmod (state\_size/8);$
    $S[i] = S[i] \oplus (S[index] << 1);$
    $S[index] = S[index] \oplus (S[i] << 3);$
end if;
```

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

```
$i1 = (i - 1) \bmod (state\_size/8);$
if ($i \bmod 4 == 3$)
    $index = S[i1] \bmod (state\_size/8);$
    $S[i] = S[i] \oplus (S[index] << 1);$
    $S[index] = S[index] \oplus (S[i] << 3);$
end if;
```

1.1.4 Implementation of the functions in C programming

The above functions can implemented efficiently in C code.

Note 1. $y = x \bmod (state\_size/8)$ is implemented as follows:

```
$mask = state\_size/8 - 1$ ;
$y = x \,\&\, mask$ ;
```

Note 2. In Step 4 of Sect. 1.2, $temp = 5^{j \times (state\_size/8)+i}$ is implemented as:

```
$temp = 1;$
for $j = 0$ to $2^{t\_cost} - 1$ do
    for $i = 0$ to $(state\_size/8) - 1$, do
        $temp = temp + (temp << 2);$    // i.e., temp = temp*5
    end for;
end for;
```

1.2 Hashing the password and salt

The hashing algorithm is given below. Note that $S[i]$ is a 64-bit unsigned integer, and $S[i] = S_{8i} \parallel S_{8i+1} \parallel S_{8i+2} \parallel S_{8i+3} \parallel S_{8i+4} \parallel S_{8i+5} \parallel S_{8i+6} \parallel S_{8i+7}$.

```
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 $S_i = pwd_i$ for $i = 0$ to $pwd\_size - 1$;
   Let $S_{128+i} = salt_i$ for $i = 0$ to $salt\_size - 1$;
   Let $S_{160} = pwd\_size$;
   Let $S_{161} = salt\_size$;
   Let $S_{162} = output\_size$;

3. // expand the data into the whole state.
   for $i = 41$ to $(state\_size/8) - 1$, do: $\mathrm{F}(S, i)$;

4. // update the state using F and G
   // (involving deterministic random memory accesses)
   for $j = 0$ to $2^{t\_cost} - 1$ do
       for $i = 0$ to $(state\_size/8) - 1$, do
           $\mathrm{F}(S, i)$; $\mathrm{G}(S, i, j)$;
       end for;
   end for;

5. // update the state using F
   for $i = 0$ to $(state\_size/8) - 1$, do: $\mathrm{F}(S, i)$;

6. // update the state using F and H
   // (involving password-dependent random memory accesses)
   for $j = 0$ to $2^{t\_cost} - 1$ do
       for $i = 0$ to $(state\_size/8) - 1$, do
           $\mathrm{F}(S, i)$; $\mathrm{H}(S, i)$;
       end for;
   end for;

7. // update the state using F
   for $i = 0$ to $(state\_size/8) - 1$, do: $\mathrm{F}(S, i)$;

8. The hash output is given as the last $t$ bytes of the state $S$ ($t <= 128$):
   $S_{state\_size-t} \parallel S_{state\_size-t+1} \parallel \cdots \parallel S_{state\_size-1}$ .
```

1.3 Recommended Parameters

The memory size of POMELO is 2^{13+m_cost} bytes. The number of steps is about $2^{11+m\_cost+t\_cost}$ (in each step, one element of the state gets updated). The number of random memory accesses (one access includes one read and one write) is $2^{9+m\_cost+t\_cost}$.

We recommend $8 \leq m\_cost + t\_cost \leq 20$. When $m\_cost + t\_cost = 8$, it is very fast to compute the password hashing. When $m\_cost + t\_cost = 20$, we get very high security, but it is also expensive for the defender 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 Sect. 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, while still wants to use large memory in the computation, the user may use $m\_cost = 12$, $t\_cost = 1$ (2^{25} bytes of memory).

Chapter 2

Initial 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. In another words, POMELO is one-way.

Claim 2. POMELO is strong against the attacks that intend to bypass the large memory in POMELO. The reason is that we used functions G(S, i, j) and H(S, i) which read and write at random memory locations.

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 since the first half of POMELO uses deterministic memory accesses.

2.1 Preimage/Collision Security

For any password hashing algorithm, preimage/collision security is exected to be easily achievable, since in general we can perform a lot of computations and use large state in the algorithm.

In POMELO, in the password expansion stage (Step 3 in Sect. 1.2), it expands the password and salt into a state with size at least 8192 bytes through an invertible non-linear function. At this stage, we can expect that POMELO already has strong collision security.

In POMELO, the output is taken from the last t bytes of the state after updating the state in Steps 4, 5, 6, and 7. The simple extraction is sufficient to prevent the input being retrieved from the output faster than guessing the inputs.

2.2 Low Memory Attack

An important design goal of password hashing algorithm is to prevent the large memory state being effectively bypassed in password cracking, as achieved in both bcrypt [2] and scrypt [1].

In POMELO, the functions G and H provide effective way against the attacks that try to use smaller memory. The reason is that both G and H accesses memory in a random pattern (with read and write). Writing to random memory locations makes it much harder to use a small part of the state data to reconstruct the state efficiently.

Function G is not as strong as function H, since the memory access in G is deterministic, while the memory access in H is password-dependent. But the advantage of using G is that it remains secure against the cache-timing attack.

2.3 SIMD Attack

The SIMD is efficient since the instruction decoding circuits can be greatly saved. In POMELO, the H prevents completely the attack using SIMD 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 inforamtion 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, the large memory in POMELO makes the cost-saving of instruction decoding in SIMD much less effective, comparing to the memory cost. So we believe that POMELO provides strong security against SIMD attack.

Chapter 3

Efficiency Analysis

3.1 Software Performance

The state size of POMELO is given as 2^{13+m_cost} bytes, and the computational cost of POMELO is mainly dominated by those $2^{9+m\_cost+t\_cost}$ random memory accesses on the mainframe computers (when the state size is much larger than the memory cache size).

We implemented POMELO in C code. We tested the speed on Lenovo X220 with Intel Core i5-2540M 2.6GHz processor (maximum speed of 3.3GHz) running 64-bit Ubuntu 11.04. The RAM size is 4 Gigabytes. The compiler being used is gcc 4.5.2, and the optimization option "-O3" is used. The code being tested is the one submitted to PHC.

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 Gigabyte, it takes 9.3 seconds ($m\_cost = 17$, $t\_cost = 0$) to hash a password. When the state size of POMELO is 32 Megabyte, it takes only 0.15 seconds ($m\_cost = 12$, $t\_cost = 0$) to hash a password.

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 password 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 impossible to utilize the SIMD power of GPU to crack passwords protected by POMELO.

However, in case 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

Table 3.1: The timing of POMELO on Intel Core i5-2540M

m_cost	t_cost	state size (bytes)	timing (seconds)
7	0	2^{20}	0.0013
8	0	2^{21}	0.003
9	0	2^{22}	0.011
10	0	2^{23}	0.03
11	0	2^{24}	0.07
12	0	2^{25}	0.15
13	0	2^{26}	0.32
14	0	2^{27}	0.69
15	0	2^{28}	1.6
16	0	2^{29}	3.9
17	0	2^{30}	9.3
7	10	2^{20}	1.3
8	9	2^{21}	1.5
9	8	2^{22}	2.7
10	7	2^{23}	3.9
11	6	2^{24}	4.5
12	5	2^{25}	4.9
13	4	2^{26}	5.2
14	3	2^{27}	5.5
15	2	2^{28}	6.4
16	1	2^{29}	7.8
17	0	2^{30}	9.3

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 output being generated from the algoritm with the old parameters.

Chapter 4

Features

- Simple design. POMELO is based on a simple non-linear feedback function F, and two simple random memory access functions G and H. It does not use any existing hash function or cipher in the design.
- Easy to implement. The algorithm description can be easily translated into programming code.
- Adjustable memory size and computational complexity.
- Efficient. In POMELO, large memory can be used, and the computation can still be efficient on the main frame computers.
- Deterministic random memory accesses (read/write) are used in the first half of POMELO. It is resistant to the cache-timing attack. It is difficult to bypass the memory requirement due to the random memory accesses.
- Password-dependent random memory accesses are used in the second half of POMELO. It prevents SIMD completely. In case that the cache-timing attack is successful against this part, the password being protected by POMELO still remains secure since the attacker has to attack the first half of POMELO in order to recover the password.

Chapter 5

No hidden weakness

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

Chapter 6

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

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