Performance Analysis of Some Password Hashing Schemes

Donghoon Chang, Arpan Jati, Sweta Mishra, Somitra Kumar Sanadhya

February 20, 2015

Abstract

In this work we have analyzed some password hashing schemes for performance under various settings of time and memory complexities. We have attempted to benchmark the said algorithms at similar levels of memory consumption. Given the wide variations in security margins of the algorithms and incompatibility of memory and time cost settings, we have attempted to be as fair as possible in choosing the various parameters while executing the benchmarks.

Keywords: Password hashing, benchmark, PHC

1 Benchmarking setup

In order to get consistent results for the different algorithms we performed all the test on a single machine with the code compiled by the same compiler. The details are as follows:

- **CPU:** Intel Core i7 4770 (Turbo Boost: ON) - Working at 3.9 GHz
- **RAM:** Double Channel DDR3 16 GB (2400 MHz)
- **Compiler:** gcc / g++ v4.9.2 (-march=native and -O3 flags were set if not already in the makefiles). This would cause the compiler to use the AVX-2 instructions.
- **OS:** UBUNTU 14.04.1, on HYPER-V, on Windows-8.1 with 8 GB allocated RAM to the VM. We also performed benchmarks on native Linux OS to make sure that the virtualization does not cause any changes in the results.

2 Performance Analysis

In this section we provide the benchmarking results and details of the setup and considerations.

For consistency we used single threaded versions of the algorithms.

All the experiments were run at-least 5 times and the average timings were taken.

2.1 Catena v3 - Butterfly and Dragonfly [6]

The Catena v3 document provides the two new variants called Dragonfly and Butterfly based on the instantiation of Catena-BRG and Catena-DBG. Earlier versions were significantly slower than the current version with the single round Blake2b-1 function.

Catena v3 Butterfly (at different values of lambda with BLAKE2b-1)					
t_cost	128 MB (m_cost=20)	256 M (m_cost=21)	512 MB (m_cost=22)	1024 MB (m_cost=23)	2048 MB (m_cost=24)
lambda=1	1.48 sec 86.2 MB/s	3.03 sec 84 MB/s	6.45 sec 79.35 MB/s	13.4 sec 74.0 MB/s	28.0 sec 73.1 MB/s
lambda=2	2.9 sec 44 MB/s	6.00 sec 42.6 MB/s	12.6 sec 40.3 MB/s	26.9 sec 38.0 MB/s	58.1 sec 36.5 MB/s
lambda=3	4.3 sec 29 MB/s	9.10 sec 28 MB/s	19.4 sec 26.2 MB/s	40.0 sec 25.3 MB/s	83.3 sec 24.6 MB/s

Table 1: Catena v3 - Butterfly

The Catena-Dragonfly is much faster than Catena-Butterfly, but, the Dragonfly version is shown not to be memory-hard in [4], [2].

The latest version of code at the time of benchmark was cloned from 'https://github.com/cforler/catena/' (75 commits).

Optimized SSE implementation was used for both the benchmarks.

Catena v3 Dragonfly (at different values of lambda with BLAKE2b-1)					
t_cost	128 MB	256 MB	512 MB	1024 MB	2048 MB
lambda=1	0.237 sec 539 MB/s	0.495 sec 516 MB/s	0.989 sec 517 MB/s	1.96 sec 520 MB/s	4.0 sec 511 MB/s
lambda=2	0.29 sec 437 MB/s	0.592 sec 431 MB/s	1.205 sec 424 MB/s	2.374 sec 431 MB/s	4.855 sec 421 MB/s
lambda=3	0.46 sec 273 MB/s	0.954 sec 268 MB/s	1.932 sec 264 MB/s	3.834 sec 267 MB/s	7.837 sec 261 MB/s
lambda=4	0.534 sec 239 MB/s	1.073 sec 238 MB/s	2.146 sec 238 MB/s	4.288 sec 238 MB/s	8.694 sec 235 MB/s

Table 2: Catena Dragonfly

One of the reasons for the slow nature of the Butterfly version is the need for $2{\cdot}g$ rows for processing. This property of Catena-DBG, combined with the relatively small read-writes to the RAM makes the overall structure significantly slow. Even the fastest version of Catena-Butterfly-Blake2b-1 can only achieve overall memory hashing speed of around 80 MiB/s.

The Catena-Dragonfly (Table 2) is much faster due to the significantly reduced number of rounds as compared to Catena-Butterfly (Table 1). Also, the way it operates, every node in the Catena-BRG graph has dependency on two previous ancestors as opposed to three of Catena-DBG. This leads to reduced number of random memory accesses and faster speeds.

2.2 Gambit[10]

For the benchmarking of Gambit we used v1 of the source code from [1]. No optimized version of the code was available, and we used the reference implementation for this analysis. The speeds are particularly slow due to the slow performance of Keccak sponge in software and small memory access chunks. One peculiarity of Gambit is that the Time Cost and Memory Cost is bound by the assertion ($cost\_m \times 2 \leq cost\_t \times (r/8)$) and $r = 136$ for $Gambit - 256$. For the benchmark we set *t_cost* to the lowest possible value for a required *m_cost* for fixed memory consumption and defined it as $t = 1$, for higher values of t we doubled the *t_cost* for every subsequent values of t. This was done to have a consistent range of possible speeds with increasing t. Results are shown in Table 3.

Gambit-v1	Memory processing rate of Gambit-256			
t_cost	128 MB	256 MB	512 MB	1024 MB
t=1 1597831	1.22 sec 104.7 MB/s	2.47 sec 103 MB/s	4.84 sec 105.6 MB/s	9.57 sec 106.9 MB/s
t=2 3195662	2.38 sec 53.63 MB/s	4.93 sec 51.84 MB/s	10.43 sec 49.06 MB/s	19.62 sec 52.18 MB/s
t=3 4793493	3.67 sec 34.78 MB/s	7.46 sec 34.29 MB/s	14.65 sec 34.9 2MB/s	28.75 sec 35.6 MB/s
t=4 6391324	4.82 sec 26.54 MB/s	9.82 sec 26 MB/s	19.11 sec 26.79 MB/s	38.14 sec 26.84 MB/s

Note: t_cost values are for 128 MB.

Table 3: Gambit v1

2.3 Lyra2 - v3[7]

For benchmarking Lyra-2 we used the v3 code from [1]. We used the SSE version of Lyra2-v3 with nPARALLEL=1 for consistency. We noticed that the speed of Lyra-2 with multiple threads is faster than the single thread ones (as expected), but for consistency, we choose to use the single threaded version. The default Makefile was used with linux-x86-64-sse, with nThreads=1. This would result in fast AVX implementation being used with Blake2b as sponge function. We did receive a warning for large-memory-usage for the 2 GiB test, but, the timings are as expected.

Results are shown in Table 4.

Lyra2 - v3 (at different values of t and p=1)						
t_cost	200 MB	400 MB	800 MB	1024 MB	1600 MB	2048 MB
t=1	0.200 sec 1000 MB/s	0.372 sec 1075 MB/s	0.52 sec 1538 MB/s	0.62 sec 1651 MB/s	0.98 sec 1627 MB/s	1.20 sec 1706 MB/s
t=2	0.250 sec 800 MB/s	0.48 sec 833 MB/s	0.81 sec 987 MB/s	0.99 sec 1034 MB/s	1.47 sec 1088 MB/s	1.78 sec 1150 MB/s
t=3	0.270 sec 740 MB/s	0.51 sec 784 MB/s	0.98 sec 816 MB/s	1.26 sec 812 MB/s	1.91 sec 837 MB/s	2.40 sec 853 MB/s
t=4	0.397 sec 503 MB/s	0.708 sec 564 MB/s	1.23 sec 648 MB/s	1.57 sec 651 MB/s	2.38 sec 669 MB/s	3.00 sec 682 MB/s

Table 4: Lyra2 - v3

2.4 Rig v2[3]

For this work we used the latest version of Rig from 'https://github.com/arpanj/Rig'. We used the optimized implementation with the Blake2b round using AVX-2.

RIG v2 (BlakeExpand, BlakePerm, Blake2b) AVX-2 x86-64					
m =>	13 (128 M)	14 (256 M)	15 (512 M)	16 (1024 M)	17 (2048 M)
n = 1	0.065 sec 1966 MB/s	0.127 sec 2007 MB/s	0.259 sec 1971 MB/s	0.519 sec 1973 MB/s	1.035 sec 1978 MB/s
n = 2	0.091 sec 1405 MB/s	0.181 sec 1410 MB/s	0.360 sec 1421 MB/s	0.718 sec 1425 MB/s	1.442 sec 1422 MB/s
n = 3	0.122 sec 1045 MB/s	0.243 sec 1050 MB/s	0.474 sec 1079 MB/s	0.947 sec 1081 MB/s	1.903 sec 1076 MB/s
n = 4	0.144 sec 883 MB/s	0.297 sec 861 MB/s	0.588 sec 870 MB/s	1.168 sec 876 MB/s	2.295 sec 892 MB/s

Table 5: RIG v2

All default settings were used as described in the code and Makefile. One source code improvement was the removal of writing of the data back to the memory in the last row, this change resulted in around five percent improvement in overall performance for small values of N.

Results are shown in Table 5.

2.5 Scrypt[8]

Scrypt is the first memory-hard algorithm for password-hashing. There are several implementations of Scrypt available, we used one of the fastest variants of the implementation by @floodyberry in this work. Table 6 shows the results of the AVX2 implementation with Blake2b and Salsa64/8.

Scrypt @floodyberry's https://github.com/floodyberry/scrypt-jane

Scrypt (AVX2, Blake2b, Salsa 64/8, x86-64)		
Memory (MB)	Time (second)	Speed (MB/s)
128	0.076	1684
256	0.162	1580
512	0.332	1542
1024	0.7	1530

Table 6: scrypt: floodyberry/scrypt-jane

2.6 TwoCats[5]

TwoCats is one of the fastest and one of the most complex entries of [1]. It is highly optimized to use the CPU and memory subsystem to the full extent by having several modes and multi-threading support. For the purpose of this analysis, we used the single threaded mode of TwoCats with Blake2b compiled with AVX2 support. Defaults were used among them MULTIPLIES=2, LANES=8, BLOCKSIZE=16384 and PARALLELISM=1.

The *t_cost* parameter of TwoCats is quite sensitive as it increases the iteration count of the number of small writes in cache using 2^{t_cost}.

The results are shown in Table 7.

2.7 yescrypt [9]

For benchmarking yescrypt, we used version 0.7.1 of the code from [1]. Yescrypt is another fast and complex submission to the PHC. There are several modes and settings available. For this work we used the default configuration (r=8, p=1 and YESCRYPT_RW=1). 64 bit version was used with -march=native in gcc (which essentially would have enabled AVX2 intrinsic support). The code however uses 128 bit intrinsics, so SSE4.1 should be enough to compile and enable SIMD optimizations.

The results are shown in Table 8.

TwoCats v0 (at different values of t with Blake2b)					
t_cost	128 MB	256 MB	512 MB	1024 MB	2048 MB
t=0	0.063 sec 2027 MB/s	0.136 sec 1873 MB/s	0.251 sec 2033 MB/s	0.513 sec 1995 MB/s	1.025 sec 1997 MB/s
t=1	0.115 sec 1111 MB/s	0.229 sec 1116 MB/s	0.459 sec 1115 MB/s	0.917 sec 1115 MB/s	1.842 sec 1111 MB/s
t=2	0.218 sec 586 MB/s	0.436 sec 586 MB/s	0.871 sec 587 MB/s	1.754 sec 583 MB/s	3.514 sec 582 MB/s
t=3	0.418 sec 305 MB/s	0.849 sec 301 MB/s	1.705 sec 300 MB/s	3.378 sec 303 MB/s	6.810 sec 300 MB/s
t=4	0.834 sec 153 MB/s	1.664 sec 153 MB/s	3.329 sec 153 MB/s	6.696 sec 153 MB/s	13.49 sec 152 MB/s

Table 7: TwoCats v0

yescrypt v1 (at SHA256, r=8, p=1 and YESCRYPT_RW=1)					
t_cost	128 MB	256 MB	512 MB	1024 MB	2048 MB
t=0	0.9 sec 1422 MB/s	0.180 sec 1422 MB/s	0.368 sec 1391 MB/s	0.720 sec 1422 MB/s	1.46 sec 1402 MB/s
t=1	0.110 sec 1163 MB/s	0.222 sec 1153 MB/s	0.448 sec 1143 MB/s	0.898 sec 1140 MB/s	1.782 sec 1149 MB/s
t=2	0.132 sec 969 MB/s	0.272 sec 941 MB/s	0.544 sec 941 MB/s	1.05 sec 975 MB/s	2.1 sec 975 MB/s
t=3	0.192 sec 666 MB/s	0.394 sec 649 MB/s	0.786 sec 651 MB/s	1.576 sec 649 MB/s	3.15 sec 650 MB/s
t=4	0.286 sec 447 MB/s	0.510 sec 501 MB/s	1.016 sec 503 MB/s	2.052 sec 499 MB/s	4.146 sec 493 MB/s

Table 8: Yescrypt v1

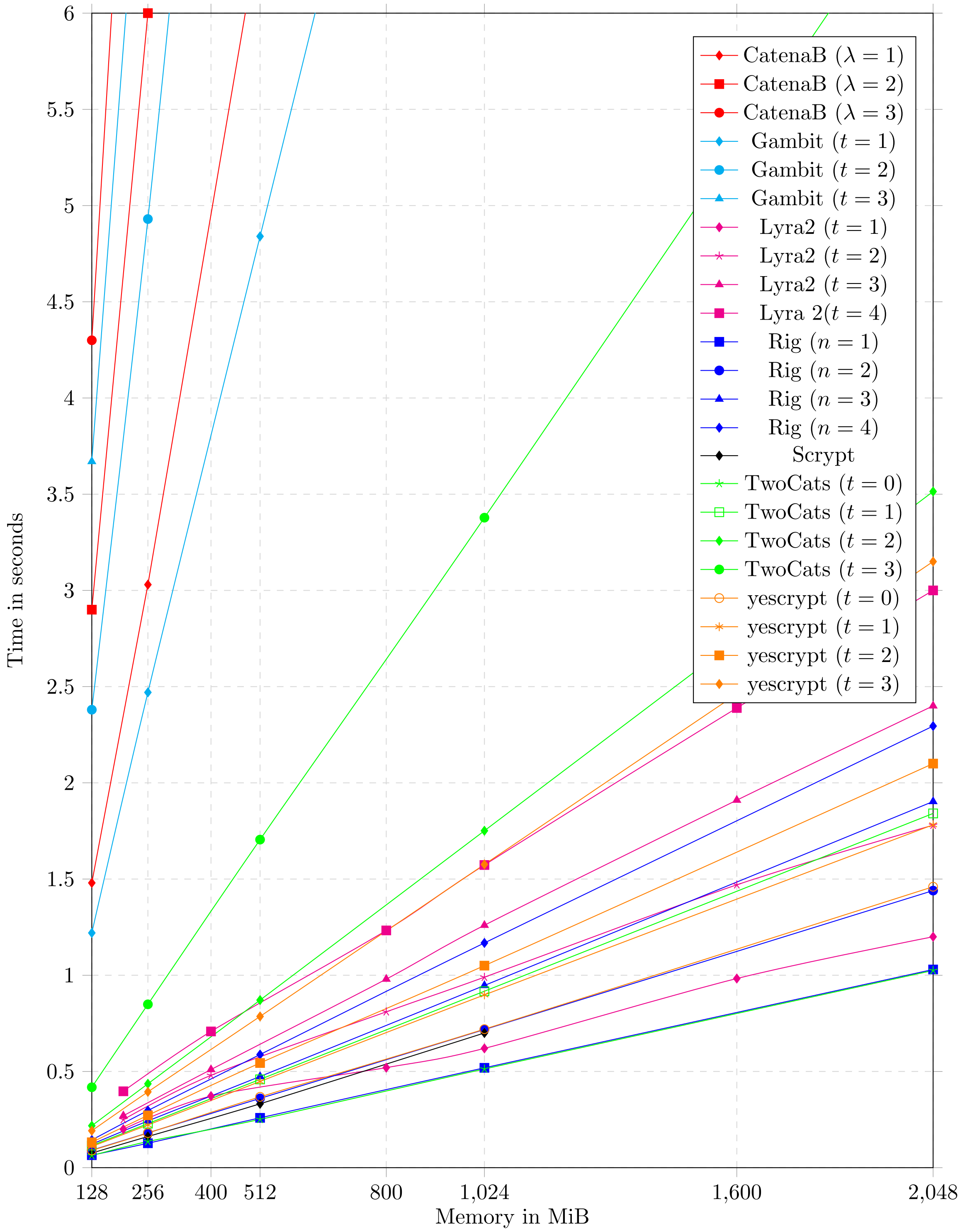

Figure 1: Performance: Memory vs. Time of some memory-hard PHC candidates

3 Conclusions

The performance graph in figure 1 shows the execution time vs. memory for all the memory-hard algorithms benchmarked. It is clear that Gambit (reference code) and Catena-Butterfly are among the slowest and take significant amount of time in hashing passwords with moderate to large amounts of memory. The performance of Gambit may be improved using a better implementation, but, the performance of Catena is unlikely to significantly improve even with native assembly implementation.

As noted before, the time cost of TwoCats is very sensitive, it may be changed with some minor tweaks to allow for better tradeoff control.

Lyra2, Rigv2, TwoCats and yescrypt provide good performance in a wide range of use cases. No, attacks are currently known against them which reduce the claimed TMTO defense.

As far as side-channels are concerned, Catena, Gambit and Rig are fully resistant; Lyra2 and TwoCats are partially resistant whereas Scrypt and yescrypt are not resistant.

References

[1] Password Hashing Competition (PHC), 2014. https://password-hashing.net/index.html.

[2] Alex Biryukov and Dmitry Khovratovich. Tradeoff Cryptanalysis of Memory-Hard Functions, 2015. http://orbilu.uni.lu/handle/10993/20043.

[3] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for Password Hashing. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/RIG-v2.pdf.

[4] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Time memory tradeoff analysis of graphs in password hashing constructions. *Preproceedings of PASSWORDS'14*, pages 256–266, 2015. http://passwords14.item.ntnu.no/Preproceedings_Passwords14.pdf.

[5] Bill Cox. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/TwoCats-v0.pdf.

[6] Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. Submission to Password Hashing Competition (PHC), 2015. https://password-hashing.net/submissions/specs/Catena-v3.pdf.

[7] Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide. Submission to Password Hashing Competition (PHC), 2015. https://password-hashing.net/submissions/specs/Lyra2-v3.pdf.

[8] Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCon*, 2009. http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.

[9] Alexander Peslyak. yescrypt - a Password Hashing Competition submission. Submission to Password Hashing Competition (PHC), 2015. https://passwordhashing.net/submissions/specs/yescrypt-v1.pdf.

[10] Krisztián Pintér. Gambit - A sponge based, memory hard key derivation function. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Gambit-v1.pdf.