# yescrypt - a Password Hashing Competition submission

| | |
|---|---|
| **Name of the submitted scheme**: | yescrypt |
| **Name and email address of the submitter(s)**: | Alexander Peslyak <solar at openwall.com> |
| **Date**: | March 31, 2014 |

# Contents



# Specification

Except for its scrypt compatibility mode, yescrypt is a work-in-progress and thus is subject to change.

## scrypt

scrypt is specified in Colin Percival's paper "Stronger Key Derivation via Sequential Memory-Hard Functions", presented at BSDCan'09 (2009):

http://www.tarsnap.com/scrypt.html

Additional commentary on scrypt is available in Colin Percival's slides from Passwords^12 (2012):

http://passwords12.at.ifi.uio.no/Colin_Percival_scrypt_Passwords12.pdf

## yescrypt

yescrypt builds upon scrypt, but is not endorsed by Colin Percival. It is currently most precisely specified by means of a deliberately mostly not optimized reference implementation found in yescrypt-ref.c. (Also provided are two optimized implementations.) In case of any incomplete or unclear or contradictory information in this document, the reference implementation is to be considered more authoritative.

Described below are the differences of yescrypt from scrypt.

### Valid values for N

Although all implementations of scrypt require that N be a power of 2, scrypt was formally defined for other values of N as well, which would rely on arbitrary modulo division. Unfortunately, because of an oversight in the scrypt specification, which was later identified in a discussion on the scrypt mailing list, such modulo division would be cumbersome or/and inefficient, since it'd have to be applied to a "big integer" value. It would also likely be variable-time (which is an added security risk) unless special care is taken to prevent that.

yescrypt, including in scrypt compatibility mode, is defined only for values of N that are powers of 2 (and larger than 1, which matches scrypt's requirements).

### Extra tunable parameters

**shared** An optional read-only lookup table (a ROM) and its access frequency mask. (The name "shared" stems from this data structure being suitable for sharing it between multiple threads, in contrast with "local", which is another data structure that exists in yescrypt's recommended MT-safe API. "local" is only a peculiarity of the API and has no effect on how yescrypt is specified.)

**flags** A bitmask allowing to enable the individual extra features of yescrypt. Currently defined are YESCRYPT_RW, YESCRYPT_PARALLEL_SMIX, and YESCRYPT_PWXFORM, corresponding to the numeric values 1, 2, and 4 (so that any combination of these flags is possible). Also defined is YESCRYPT_WORM, but it has a numeric value of 0 and is thus used to request scrypt's usual behavior when no flags are set.

**t** A 32-bit number controlling yescrypt's computation time while keeping its peak memory usage the same. t = 0 is optimal in terms of achieving the highest normalized area-time cost for ASIC attackers.

Classic scrypt is available by setting t = 0 and flags to YESCRYPT_WORM and not providing a ROM. (In the recommended API, the latter is currently achieved by passing a dummy shared structure, as described in comments in yescrypt.h, or simply by invoking the crypto_scrypt() wrapper function, which has exactly the same C prototype as in Colin Percival's implementations of scrypt.)

### ROMix algorithm changes

yescrypt supports an optional pre-filled read-only lookup table (a ROM), which it uses along with scrypt's usual sequential-write, random-read lookup table (a RAM), although this behavior is further modified when the YESCRYPT_RW flag is set (as described below). This is the "smarter" variety of the "best of both worlds" approach described in "New developments in password hashing: ROM-port-hard functions", presented at ZeroNights 2012:

http://www.openwall.com/presentations/ZeroNights2012-New-In-Password-Hashing/

When a ROM is provided, some of SMix's random reads are made from the ROM instead of from RAM. On top of that, providing a ROM and/or setting the YESCRYPT_RW flag introduces additional random reads, from the ROM and/or from RAM, beyond those that classic scrypt performed. Moreover, setting the YESCRYPT_RW flag introduces additional random writes into RAM, which classic scrypt did not perform at all.

Specifically, ROMix algorithm's steps 2 to 9 are changed from:

```
2:    for i = 0 to N - 1 do
3:      V_i ← X
4:      X ← H(X)
5:    end for
6:    for i = 0 to N - 1 do
7:      j ← Integerify(X) mod N
8:      X ← H(X ⊕ V_j)
9:    end for
```

to:

```
2:    for i = 0 to N - 1 do
3:      V_i ← X
        if ((i ∧ mask) = 1)
          j ← Integerify(X) mod NROM
          X ← X ⊕ VROM_j
        else if ((YESCRYPT_RW flag is set) and (i > 1))
          j ← Wrap(Integerify(X), i)
          X ← X ⊕ V_j
        end if
4:      X ← H(X)
5:    end for
      if (mask ≠ 0)
        mask ← mask ∨ 1
      end if
6:    for i = 0 to Nloop - 1 do
        if ((i ∧ mask) = 1)
          j ← Integerify(X) mod NROM
          X ← X ⊕ VROM_j
        else
7:        j ← Integerify(X) mod N
8.1:      X ← X ⊕ V_j
          if (YESCRYPT_RW flag is set)
            V_j ← X
          end if
        end if
8.2:    X ← H(X)
9:    end for
```

where mask is a tunable parameter controlling ROM access frequency (as well as whether to use the ROM at all), NROM is the size of the ROM in 128*r byte blocks (NROM must be a power of 2 greater than 1, just like N), VROM is the ROM lookup table indexed by block number, and Nloop is either equal to N (such as when computing classic scrypt hashes) or is an even value derived from N, t, and flags.

The Wrap() function is defined as follows:

$Wrap(x,i) = (x \bmod p2floor(i)) + (i - p2floor(i))$

where p2floor(i) is the largest power of 2 not greater than argument:

$p2floor(i) = 2^{\lfloor \log_2 i \rfloor}$

Both p2floor() and Wrap() are implementable with a handful of bitmasks, subtractions, and one addition (like it's done in yescrypt-ref.c), or the p2floor(i) value may be updated in the "for i" loop at even lower cost (like it's done in the optimized implementations).

When there's no ROM, mask should be 0. When the ROM is in the machine's RAM (same type of memory that is holding scrypt's usual V), mask should typically be set to 1, which interleaves reads from ROM and RAM in equal proportion (throughout the entire computation if YESCRYPT_RW is set, or with reads from RAM only in steps 6 to 9 if this flag is not set). When the ROM is in slower memory, such as memory-mapped from a solid-state drive (SSD), higher values of mask may be used to make ROM accesses less frequent. Even values of mask may be used to prevent ROM accesses from the first loop above, to mitigate possible security impact of SSDs' per-block read counters (please refer to the ZeroNights presentation).

It can be said that setting the YESCRYPT_RW flag changes ROMix' usage of RAM from "write once, read many" (it's "many" since in steps 6 to 9 any block in V can potentially be read from more than once) to "read-write", hence the YESCRYPT_WORM and YESCRYPT_RW names.

It is important to note that the YESCRYPT_RW flag is usable (and is almost always beneficial to use) regardless of whether a ROM is in use or not.

Setting the YESCRYPT_PARALLEL_SMIX flag has additional effect on ROMix, described in the "Thread-level parallelism" section.

#### Tunable computation time

As briefly mentioned above, yescrypt's computation time may be increased while keeping its peak memory usage the same. This is achieved via the t parameter, which in turn affects Nloop in the algorithm above. Nloop is also affected by whether the YESCRYPT_RW flag is set or not. This is in order to make t = 0 optimal in terms of achieving the highest normalized area-time cost for ASIC attackers in either case. (The optimal Nloop turned out to be different depending on YESCRYPT_RW.)

Here's how Nloop is derived from t and flags:

| **t** | **Nloop** | |
|---|---|---|
| **t** | **YESCRYPT_RW** | **YESCRYPT_WORM** |
| 0 | (N + 2) / 3 | N |
| 1 | (2N + 2) / 3 | N + (N + 1) / 2 |
| t>1 | (t-1)N | tN |

Additionally, Nloop is rounded up to the next even number (if it isn't even already), which is helpful for optimized implementations.

Here's the effect t and flags have on total computation time (including ROMix' first loop) and on area-time, both relative to classic scrypt, and on efficiency in terms of normalized area-time relative to what's optimal for the given flags settings (*not* relative to classic scrypt, which would be e.g. 300% for YESCRYPT_RW at t = 0):

| **t** | **YESCRYPT_RW** | | | **YESCRYPT_WORM** | | |
|---|---|---|---|---|---|---|
| **t** | **time** | **AT** | **ATnorm** | **time** | **AT** | **ATnorm** |
| 0 | 2/3 | 4/3 | 100% | 1 | 1 | 100% |
| 1 | 5/6 | 2 | 96% | 1.25 | 1.5 | 96% |
| 2 | 1 | 8/3 | 89% | 1.5 | 2 | 89% |
| 3 | 1.5 | 14/3 | 69% | 2 | 3 | 75% |
| 4 | 2 | 20/3 | 56% | 2.5 | 4 | 64% |
| 5 | 2.5 | 26/3 | 46% | 3 | 5 | 56% |

The area-time costs for YESCRYPT_RW given in this table, relative to those of classic scrypt, are under assumption that YESCRYPT_RW is fully effective at preventing TMTO from reducing the area-time, whereas it

is well-known that classic scrypt's TMTO allows not only for the tradeoff, but also for a decrease of attacker's area-time by a factor of 2 for ROMix' second loop (and far worse for the first loop, which was not even considered in the original scrypt attack cost estimates). In case this assumption does not hold true, YESCRYPT_RW's relative area-time costs may theoretically be up to twice lower than those shown above (but for t = 0 they would still be at least 1.5 times higher than classic scrypt's assuming that rN is scaled up to achieve same computation time). Note that this is not an assumption that YESCRYPT_RW is effective at making TMTO infeasible for the purpose of trading time for memory (although this is probably true as well), but merely that there's no longer a decrease in area-time product from whatever TMTO attacks there may be.

Since t = 0 is optimal in terms of achieving the highest normalized area-time cost for ASIC attackers, higher computation time, if affordable, is best achieved by increasing N rather than by increasing t. However, if the higher memory usage (which goes along with higher N) is not affordable, or if fine-tuning of the time is needed (recall that N must be a power of 2), then t = 1 or above may be used to increase time while staying at the same peak memory usage. t = 1 increases the time by 25% and as a side-effect decreases the normalized area-time to 96% of optimal. (Of course, in absolute terms the area-time increases with higher t. It's just that it would increase slightly more with higher rN rather than with higher t.) t = 2 increases the time by another 20% and decreases the normalized area-time to 89% of optimal. Thus, these two values are reasonable to use for fine-tuning. Values of t higher than 2 result in further increase in time while reducing the efficiency much further (e.g., down to around 50% of optimal for t = 5, which runs 3.75 or 3 times slower than t = 0, with exact numbers varying by the flags settings).

**Thread-level parallelism**

In classic scrypt, setting p > 1 introduces parallelism at (almost) the highest level. This has the advantage of needing to synchronize the threads just once (before the final PBKDF2), but it results in greater flexibility for both the defender and the attacker, which has both pros and cons: they can choose between sequential computation in less memory (and more time) and parallel computation in more memory (and less time) and various in-between combinations.

The YESCRYPT_PARALLEL_SMIX flag moves this parallelism to a slightly lower level, inside SMix. This reduces flexibility for efficient computation (for both attackers and defenders) by requiring that, short of resorting to a TMTO attack on ROMix, the full amount of memory be allocated as needed for the specified p, regardless of whether that parallelism is actually being fully made use of or not. This may be desirable when the defender has enough memory with sufficiently low latency and high bandwidth for efficient full parallel execution, yet the required memory size is high enough that some likely attackers might end up being forced to choose between using higher latency memory than they could use otherwise (waiting for data longer) or using TMTO (waiting for data more times per one hash computation). The area-time cost for other kinds of attackers (who would use the same memory type and TMTO factor or no TMTO either way) remains roughly the same, given the same running time for the defender. In the TMTO-friendly YESCRYPT_WORM mode, as long as the defender has enough memory that is just as fast as the smaller per-thread regions would be, doesn't expect to ever need greater flexibility (except possibly via TMTO), and doesn't need backwards compatibility with classic scrypt, there are no other serious drawbacks to this setting. In the YESCRYPT_RW mode, which is meant to discourage TMTO, this new approach to parallelization makes TMTO less inefficient. (This is an unfortunate side-effect of avoiding some random writes, as we have to in order to allow for parallel threads to access a common memory region without synchronization overhead.) Thus, in this mode this setting poses an extra tradeoff of its own (higher area-time cost for a subset of attackers vs. better TMTO resistance). Setting YESCRYPT_PARALLEL_SMIX also changes the way the running time is to be controlled from rpN (for classic scrypt) to rN (in this modification). (Of course, it may also be controlled via t, with an efficiency loss.)

All of this applies only when p > 1. For p = 1, YESCRYPT_PARALLEL_SMIX is a no-op.

To introduce YESCRYPT_PARALLEL_SMIX, the original SMix is split in two algorithms: SMix1 contains ROMix steps 1 to 5 and step 10 (excludes steps 6 to 9), and SMix2 contains ROMix step 1 and steps 6 to 10 (excludes steps 2 to 5).

A new SMix algorithm is then built on top of these two sub-algorithms:

```
1:   n ← N/p
2:   Nloop_all ← fNloop(n,t,flags)
3:   if (YESCRYPT_RW flag is set)
4:     Nloop_rw ← Nloop_all/p
5:   else
6:     Nloop_rw ← 0
7:   end if
8:   n ← n−(n mod 2)
9:   Nloop_all ← Nloop_all+(Nloop_all mod 2)
10:  Nloop_rw ← Nloop_rw−(Nloop_rw mod 2)
11:  for i=0 to p−1 do
12:    s ← in
13:    if (i=p−1)
14:      n ← N−s
15:    end if
16:    e ← s+n−1
17:    SMix1_r(B_i,n,V_s..e,flags)
18:    SMix2_r(B_i,p2floor(n),Nloop_rw,V_s..e,flags)
19:  end for
20:  for i=0 to p−1 do
21:    SMix2_r(B_i,N,Nloop_all−Nloop_rw,V,flags excluding YESCRYPT_RW)
22:  end for
```

where fNloop(n, t, flags) derives Nloop as described in the previous section, but using n in place of N and skipping the rounding up to even number (this is postponed to step 9 in the new SMix algorithm above).

In a parallelized implementation, the threads need to be synchronized between the two loops, but individual loop iterations may all proceed in parallel. (This is implemented by means of OpenMP in the provided optimized implementations.) In the first loop, the threads operate each on its own portion of V, so they may perform both reads and writes. In the second loop, they operate on the entire (shared) V, so they treat it as read-only.

When the YESCRYPT_PARALLEL_SMIX flag is not set, the new SMix algorithm is always invoked with p set to 1, which makes it behave exactly like the original SMix did. A (possibly parallel) loop for the actual p is in that case kept outside of SMix, like it is in original scrypt.

### YESCRYPT_PWXFORM

Setting the YESCRYPT_PWXFORM flag replaces most uses of Salsa20/8 with those of yescrypt's custom pwxform algorithm.

First, pwxform S-boxes are initialized with:

```
1:   for i=0 to p−1 do
2:     SMix1_1(B_i, S_SIZE_ALL /128,S_i,flags excluding YESCRYPT_PWXFORM)
3:   end for
```

They are expected to be small enough that r=1 (hard-coded in the algorithm above) is efficient for their initialization, regardless of what larger value of r may be used for the rest of computation. (If they are not small enough, we'll incur worse efficiency loss in pwxform itself anyway.) Note that these initial uses of SMix1 still use solely Salsa20/8, thereby avoiding a chicken-egg problem. Also note that they update B, and it's this updated B that is then input to SMix as defined above. (In fact, implementations may include these extra invocations of SMix1 into the first loop in the SMix algorithm defined above, thereby avoiding what would be an unnecessary thread synchronization point. The provided implementations do just that.)

Then further invocations of SMix1 and SMix2 use a variation of the BlockMix algorithm that differs from scrypt's usual one in that it doesn't shuffle output sub-blocks, uses pwxform in place of Salsa20/8 for as long as sub-blocks

processed with pwxform fit in the provided block B, and finally uses Salsa20/8 to post-process the last sub-block output by pwxform (thereby finally mixing pwxform's parallel lanes). If pwxform is tuned such that its blocks are smaller than 64 bytes, then this final Salsa20/8 invocation processes multiple such blocks accordingly. If pwxform is tuned such that its blocks are larger than 64 bytes, then Salsa20/8 is invoked multiple times accordingly, in the same fashion that scrypt's original BlockMix normally does it.

BlockMix's shuffling of output blocks appears to be unnecessary, or/and fully redundant given other data dependencies. Moreover, it is a no-op at r=1, which is a supported setting for scrypt. This topic was brought up on the scrypt mailing list, but was not discussed:

http://mail.tarsnap.com/scrypt/msg00059.html

#### Inter-mixing pwxform and Salsa20

pwxform stands for "parallel wide transformation", although it can as well be tuned to be as narrow as one 64-bit lane. It operates on 64-bit lanes. Inter-mixing pwxform with Salsa20/8, like we do in revised BlockMix described above, requires that Salsa20/8's 32-bit words fall into the same 64-bit lanes for pwxform in all implementations, running on all platforms. Unfortunately, Salsa20's most optimal data layout varies between scalar and SIMD implementations. In yescrypt, a decision has been made to favor SIMD implementations, in part because due to their speed the *relative* impact of data shuffling on them would have been higher. Thus, the Salsa20 data layout used along with pwxform is the SIMD-shuffled layout, including on scalar implementations. Additionally, for the purpose of inter-mixing with Salsa20's 32-bit words, pwxform's 64-bit words are assumed to be in little-endian order (of their 32-bit halves in this case).

### No hidden weaknesses

There are no deliberately hidden weaknesses in yescrypt.

## Initial security analysis

### Cryptographic security

Cryptographic security of yescrypt (collision resistance, preimage and second preimage resistance) is based on that of SHA-256, HMAC, and PBKDF2. The rest of processing, while crucial for increasing the cost of password cracking attacks, may be considered non-cryptographic. Even a catastrophic failure of yescrypt's SMix (and/or deeper layers) to maintain entropy would not affect yescrypt's cryptographic properties as long as SHA-256, HMAC, and PBKDF2 remain unbroken.

That said, in case SHA-256 is ever broken, yescrypt's additional processing, including its use of Salsa20/8 and more, is likely to neutralize the effect any such break.

Except in scrypt compatibility mode, improvements have been made to:

1. Avoid HMAC's and PBKDF2's trivial "collisions" that were present in classic scrypt due to the way HMAC processes the key input. (These had no security impact on intended uses of scrypt.) Specifically, a password of 65 characters or longer and its SHA-256 hash would both produce the same scrypt hash, but they do not produce the same native yescrypt hashes.

2. (By)pass not only password, but also salt entropy into the final PBKDF2 step. Thus, a potential failure of yescrypt's SMix (and/or deeper layers) will not affect yescrypt's cryptographic properties with respect not only to the password input, but also to the salt input.

# Efficiency analysis

## Defense performance

Please refer to the PERFORMANCE-* text files for yescrypt's performance figures obtained for different usage scenarios on different platforms. In summary, very decent performance is achieved in terms of hashes computed per second or the time it takes to derive a key, as well as in terms of memory bandwidth usage.

yescrypt with (at least) the YESCRYPT_PWXFORM flag set is able to exploit arbitrarily wide SIMD vectors (any number of 64-bit lanes), with or without favoring CPUs capable of gather loads, and provide any desired amount of instruction-level parallelism. In the current implementations, these parameters are tunable at compile-time (and indeed they affect the computed hashes). For a future revision of yescrypt, the intent is to make these parameters runtime tunable and to provide both generic and specialized code versions (for a handful of currently relevant sets of settings), and to encode the parameters along with computed hashes.

Just like scrypt, yescrypt is also able to exploit thread-level parallelism for computation of just one hash or derived key. Unlike in scrypt, there's an extra approach at thread-level parallelization in yescrypt, enabled via the YESCRYPT_PARALLEL_SMIX flag. (There's a lengthy description of it in the yescrypt.h file.) Two of the provided implementations (the optimized scalar and the SIMD implementation) include OpenMP support for both approaches at yescrypt's parallelism.

## Attack performance

### GPU

At small memory cost settings, yescrypt with (at least) the YESCRYPT_PWXFORM flag set discourages GPU attacks by implementing small random lookups similar to those of bcrypt. With current default settings and running the SIMD implementation on a modern x86 or x86-64 CPU (such as Intel's Sandy Bridge or better, or AMD's Bulldozer or better), yescrypt achieves frequencies of small random lookups and of groups of (potentially) parallel small random lookups that are on par with those of bcrypt. (In case of groups of (potentially) parallel lookups, the frequency is normalized for S-box size, since the relevant GPU attack uses the scarce local memory.)

bcrypt's efficiency on current GPUs is known to be extremely poor (making contemporary GPUs and CPUs roughly same speed at bcrypt per-chip), from three independent implementations. The current limiting factors are: GPUs' low local memory size (compared even to bcrypt's 4 KiB S-boxes per instance), high instruction latencies (compared to CPUs), and (for another attack) the maximum frequency of random global memory accesses (as limited by global memory bandwidth divided by cache line size).

yescrypt tries to retain bcrypt's GPU resistance while providing greater than bcrypt's (and even than scrypt's) resistance against ASICs and FPGAs. Improving upon bcrypt's GPU resistance is possible, but unfortunately it currently involves yescrypt settings that are suboptimal for modern CPUs (leaving too little parallelism to fully exploit those CPUs for defense), thereby reducing resistance against some non-GPU attacks (even attacks with CPUs, where the parallelism would be re-added from multiple candidate passwords to test at once).

At much larger memory cost settings, yescrypt with (at least) the YESCRYPT_RW flag set additionally discourages GPU attacks through discouraging time-memory tradeoffs (TMTO) and thereby limiting the number of concurrent instances that will fit in a GPU card's global memory. The more limited number of concurrent instances (compared e.g. to classic scrypt, which is TMTO-friendly) prevents the global memory access latency from being hidden or even leaves some computing resources idle all the time (like it also happens with YESCRYPT_PWXFORM above due to limited local memory).

Setting both flags at once achieves the best effect, regardless of memory cost setting.

#### ASIC and FPGA

yescrypt with (at least) the YESCRYPT_PWXFORM flag set performs rapid random lookups (as described above), typically from a CPU's L1 cache, along with 32x32 to 64-bit integer multiplications. Both of these operations have latency that is unlikely to be made much lower in specialized hardware than it is in CPUs. (This is in contrast with bitwise operations and additions found in Salsa20/8, which is the only type of computation performed by classic scrypt in its SMix and below. Those allow for major latency reduction in hardware.) For each sub-block of data processed in BlockMix, yescrypt computes multiple sequential rounds of pwxform, thereby imposing a lower bound on how quickly BlockMix can proceed, even if a given hardware platform's memory bandwidth would otherwise permit for much quicker processing.

yescrypt with (at least) the YESCRYPT_RW flag set additionally discourages time-memory tradeoffs (TMTO), thereby reducing attackers' flexibility. Perhaps more importantly, yescrypt's YESCRYPT_RW increases the area-time cost of attacks, and this higher cost of attacks is achieved at a lower (defensive) running time. Specifically, scrypt achieves its optimal area-time cost at 2*N combined iterations of the loops in SMix, whereas yescrypt achieves its optimal area-time cost at 4/3*N iterations (thus, at 2/3 of classic scrypt's running time) and, considering the 2x area-time reduction that occurs along with exploitation of TMTO in classic scrypt, that cost is higher by one third (+33%). Normalized for the same running time (which lets yescrypt use 1.5 times higher N), the area-time cost of attacks on yescrypt is 3 times higher than that on scrypt.

Like with GPU attacks, setting both flags at once achieves the best effect also against specialized hardware.

## Code

Three implementations are included: reference (mostly not optimized), somewhat optimized scalar, and heavily optimized SIMD (currently for x86 and x86-64 with SSE2, SSE4.1, AVX, and/or XOP extensions).

yescrypt's native API is provided and documented via lengthy comments in the yescrypt.h file.

The PHC mandated API is provided in the phc.c file.

Test vectors are provided in TESTS-OK (for the native API) and PHC-TEST-OK (for the PHC mandated API). Test programs are built and run against the test vectors by "make check". Please refer to the README file for more detail on this.

## Intellectual property statement

yescrypt 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.

## Thanks


- Colin Percival
- Bill Cox
- Rich Felker
- Anthony Ferrara
- Christian Forler
- Samuel Neves

- Christian Winnerlein (CodesInChaos)
- DARPA Cyber Fast Track