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>
Originally submitted on: March 31, 2014
This revision: October, 2015

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). (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. Unlike "shared", "local" is only a peculiarity of the API and is not a parameter to yescrypt.)

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.

g The number of cost upgrades performed to the hash so far. 0 means no upgrades yet.

flags A bitmask allowing to enable the individual extra features of yescrypt. Currently defined are YESCRYPT_RW (yescrypt's native mode) and YESCRYPT_WORM (conservative enhancement of classic scrypt), which can't be enabled both at once. flags = 0 requests classic scrypt. It is intended that flags will also encode pwxform parameters (which in the current implementations are specified at compile time).

Classic scrypt is available by setting $t = 0$, $g = 0$, flags = 0, and not providing a ROM. The provided implementations also include 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 (have ROM) and ((i ∧ 1) ≠ 0)
              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
6:    for i = 0 to Nloop − 1 do
          if (have ROM) and ((i ∧ 1) ≠ 0)
              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 VROM is the optional ROM lookup table of NROM blocks (128r bytes each) indexed by block number (starting with zero) and Nloop is an even value derived from N, t, and flags as specified below. NROM must be a power of 2 greater than 1, just like N.

The Wrap() function is defined as follows:

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

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).

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_RW flag has additional effect on ROMix when p > 1, as 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_RW 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.

As a side effect of differences between the algorithms, setting YESCRYPT_RW also changes the way the total processing time (combined for all threads) and memory allocation (if the parallelism is being made use of) is to be controlled from N*r*p (for classic scrypt) to N*r (in this modification). Obviously, these only differ for p > 1, and of course t takes effect as well in any case.

To introduce the above change, 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:

```
n ← N/p
Nloop_all ← fNloop(n, t, flags)
if YESCRYPT_RW flag is set
   Nloop_rw ← Nloop_all/p
else
   Nloop_rw ← 0
end if
n ← n − (n mod 2)
Nloop_all ← Nloop_all + (Nloop_all mod 2)
Nloop_rw ← Nloop_rw + (Nloop_rw mod 2)
for i = 0 to p − 1 do
   u ← in
   if i = p − 1
      n ← N − u
   end if
```

16: $v \leftarrow u+n-1$
17: if YESCRYPT_RW flag is set
18: $SMix1_1(B_i, Sbytes/128, S_i,$ no flags)
19: $S2_i \leftarrow S_{i,0\ldots 2^{Swidth}-1}$
20: $S1_i \leftarrow S_{i,2^{Swidth}\ldots 2\cdot 2^{Swidth}-1}$
21: $S0_i \leftarrow S_{i,2\cdot 2^{Swidth}\ldots 3\cdot 2^{Swidth}-1}$
22: $w_i \leftarrow 0$
23: if $i=0$
24: $passwd \leftarrow HMAC-SHA256(B_{0,2r-1}, passwd)$
25: end if
26: end if
27: $SMix1_r(B_i, n, V_{u\ldots v}, flags)$
28: $SMix2_r(B_i, p2floor(n), Nloop_{rw}, V_{u\ldots v}, flags)$
29: end for
30: for $i=0$ to $p-1$ do
31: $SMix2_r(B_i, N, Nloop_{all}-Nloop_{rw}, V, flags$ excluding YESCRYPT_RW)
32: 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_RW 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.

Instruction-level parallelism

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

First, pwxform S-boxes are initialized on step 18 of the revised SMix algorithm above, where Sbytes is as specified in the next section. (S0, S1, S2, w) initialized on steps 19 through 22 are pwxform contexts, which are maintained separately for the p potential threads.

Note that r=1 is hard-coded in these initial uses of SMix1, regardless of what larger value of r may be used for the rest of computation. (This ensures that Sbytes is always divisible by this hard-coded block size, which is 128 bytes here. It also allows for cut-down implementations of yescrypt that exclude classic scrypt support to use a simplified BlockMix, as well as for full implementations to reuse this SMix1 and its BlockMix for classic scrypt support. Using r=1 here is OK performance-wise because classic scrypt's SMix1, which is what we use here, performs sequential writes only, as well as because the S-boxes are small.)

Also note that these initial uses of SMix1 still use solely Salsa20/8, thereby avoiding a chicken-egg problem, and they also don't use the ROM even if present. (Both of these design decisions are also helpful for sharing of this SMix1 implementation with classic scrypt's.)

Finally, note that these initial uses of SMix1 update B, and it's this updated B that is then input to the main SMix1 invoked on step 27.

Then further invocations of SMix1 and SMix2 (on steps 27, 28, and 31) use a variation of the BlockMix algorithm as specified below, reading and updating the pwxform contexts.

pwxform

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 possibly grouped into wider "simple SIMD" lanes, which are in turn possibly grouped into an even wider "gather SIMD" vector.

pwxform has the following tunable parameters (currently compile-time):

PWXsimple Number of 64-bit lanes per "simple SIMD" lane (requiring only arithmetic and bitwise operations on its 64-bit elements). Must be a power of 2.

PWXgather Number of parallel "simple SIMD" lanes per "gather SIMD" vector (requiring "S-box lookups" of values as wide as a "simple SIMD" lane from PWXgather typically non-contiguous memory locations). Must be a power of 2.

PWXrounds Number of sequential rounds of pwxform's basic transformation. Must be a power of 2, plus 2 (e.g., 3, 4, 6, 10).

Swidth Number of S-box index bits, thereby controlling the size of each of pwxform's two S-boxes (in "simple SIMD" wide elements).

For convenience, we define the following derived values:

$PWXbytes = PWXgather \cdot PWXsimple \cdot 8$

$Sbytes = 3 \cdot 2^{Swidth} \cdot PWXsimple \cdot 8$

$Smask = (2^{Swidth} - 1) \cdot PWXsimple \cdot 8$

and we use S0, S1, and S2 to refer to the current pwxform invocation's three S-boxes, which occupy non-overlapping ranges of $Sbytes/3$ bytes each in S_i as shown being initialized in the SMix algorithm above.

The pwxform algorithm is as follows:

1: for $i = 0$ to $PWXrounds - 1$ do
2: for $j = 0$ to $PWXgather - 1$ do
3: $p0 \leftarrow (lo(B_{j,0}) \wedge Smask)/(PWXsimple \cdot 8)$
4: $p1 \leftarrow (hi(B_{j,0}) \wedge Smask)/(PWXsimple \cdot 8)$
5: for $k = 0$ to $PWXsimple - 1$ do
6: $B_{j,k} \leftarrow (hi(B_{j,k}) \cdot lo(B_{j,k}) + S0_{p0,k}) \oplus S1_{p1,k}$
7: end for
8: if $(i \neq 0)$ and $(i \neq PWXrounds - 1)$
9: $S2_w \leftarrow B_j$
10: $w \leftarrow w + 1$
11: end if
12: end for
13: end for
14: $(S0, S1, S2) \leftarrow (S2, S0, S1)$
15: $w \leftarrow w \mod 2^{Swidth}$

operating on a block of PWXgather "simple SIMD" lanes, with B_j corresponding to the individual "simple SIMD" lanes and $B_{j,k}$ to their individual 64-bit unsigned integer lanes. The lo() and hi() functions extract the low and high 32-bit halves, respectively, from a 64-bit word (please also refer to the next section). The multiplication on step 6 is thus 32-bit times 32-bit producing a 64-bit unsigned result, and the addition is 64-bit unsigned with possible wraparound.

The choice of mask bits is such that if the division by (PWXsimple * 8) is omitted then p0 and p1 become direct byte offsets (or they can even be made pointers, hence the naming) rather than array indices. This simplifies the addressing modes used or avoids bit shifts (depending on architecture and "simple SIMD" lane width). Additionally, on architectures with 64-bit integers or 64-bit SIMD vector elements, steps 3 and 4 may be combined into one operation using a wider pre-computed mask containing both copies of the Smask value in the correct bit positions. The provided yescrypt-opt.c and yescrypt-simd.c implementations include both of these optimizations.

Currently, the compile-time parameters are set as follows:

```
PWXsimple = 2
PWXgather = 4
PWXrounds = 6
Swidth = 8
```

which results in a total of 512 bits of parallelism and 12 KiB S-boxes. The 128-bit "simple SIMD" lanes work well on x86's SSE2 through AVX/XOP, as well as on ARM's NEON, and having 4 of them per "gather SIMD" vector roughly matches CPUs' pipelining capacity (it may be slightly excessive on CPUs with 2-way SMT, and slightly insufficient on CPUs without SMT). On AVX2, a mix of 128-bit loads from the S-boxes and 256-bit computation may be used, although testing on Haswell shows this achieve similar performance to that of pure 128-bit SIMD code, because accessing the high 128 bits of a 256-bit AVX2 register incurs a 3-cycle latency. On AVX-512, depending on implementation in specific CPUs, gather load instructions may potentially be beneficial, and the full 512-bit SIMD registers may be used for computation. This would leave no room for pipelining within one thread, but those CPUs will likely be capable of running at least 2 or 4 threads per core (as we see in current Haswell and Knights Corner, respectively).

Another property of these settings is that the parallelism of S-box lookups is twice higher than bcrypt's (8 vs. 4), but that is compensated for with their larger size (12 vs. 4 KiB) as it relates to GPU local memory attacks. The PWXrounds default of 6 was chosen so as to approach bcrypt's rate of S-box lookups when running on current high-performance x86 CPUs starting with Intel Sandy Bridge and AMD Bulldozer, as relevant to GPU global memory attacks. This setting of PWXrounds also appears nearly optimal for maximizing the worst-case attacker's area-time product when running yescrypt defensively on current CPUs, but substituting the total number of sequential pwxform rounds performed for the time factor (in place of the measured running time) so as to better match the attacker's potential. Lower values for PWXrounds tend to produce seemingly higher area-time product on CPUs, but they are potentially more susceptible to reduction of the time factor on attackers' specialized hardware.

BlockMix_pwxform

BlockMix_pwxform differs from scrypt's BlockMix 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 Salsa20/2 (that is, Salsa20 with only one double-round) 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 Salsa20/2 invocation processes multiple such blocks accordingly. If pwxform is tuned such that its blocks are larger than 64 bytes, then Salsa20/2 is invoked multiple times accordingly, in the same fashion that scrypt's original BlockMix normally does it (except that there's no output sub-block shuffling).

The BlockMix_pwxform algorithm is as follows:

$r_1 \leftarrow 128r/PWXbytes$
$X \leftarrow B'_{r_1-1}$
for $i=0$ to r_1-1 do
  if $r_1>1$
    $X \leftarrow X \oplus B'_i$
  end if
  $X \leftarrow pwxform(X)$
  $B'_i \leftarrow X$
end for
$i=\lfloor (r_1-1)\cdot PWXbytes/64 \rfloor$

$B_i \leftarrow H(B_i)$
for $i = i+1$ to $2r-1$ do
    $B_i \leftarrow H(B_i \oplus B_{i-1})$
end for

where r must be at least $\lceil PWXbytes/128 \rceil$, B' denotes the input and output vector indexed in PWXbytes sized blocks (starting with zero), B denotes the same vector indexed in 64-byte blocks (also starting with zero), and H() is Salsa20/2.

Interfacing pwxform with Salsa20

Interfacing pwxform with Salsa20, like we do in the revised BlockMix specified above, requires that Salsa20'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 higher 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, on all implementations (including scalar).

The shuffling described in this section is to be performed on each 64-byte block in B on entry to SMix1 and SMix2, as well as on each computed Salsa20 block. The reverse transformation, which we'll refer to as unshuffling, is to be performed on each 64-byte block in B' at the very end of SMix1 and SMix2, as well as on each Salsa20 input block. With SIMD implementations of Salsa20, its unshuffling and shuffling occurs naturally, but it should nevertheless be performed explicitly on entry to and at the end of SMix1 and SMix2. (SIMD implementations of classic scrypt similarly explicitly perform shuffling and unshuffling on entry to and at the end of SMix. We simply make this data layout standard for all implementations, not just SIMD.)

The SIMD shuffle algorithm operating on a block of 16 32-bit elements (64 bytes) is as follows:

for $i = 0$ to 15 do
    $Bshuf_i \leftarrow B_{5i \mod 16}$
end for

Conversely, the SIMD unshuffle algorithm is:

for $i = 0$ to 15 do
    $B_{5i \mod 16} \leftarrow Bshuf_i$
end for

(These two algorithms are exactly the same as in SIMD implementations of classic scrypt. For classic scrypt, they are a SIMD implementation detail. For yescrypt, they're part of the specification.)

Further, we define Integerify() as extracting a 64-bit value:

$Integerify(B, r) = (B_{2r-1,1} \ll 32) \vee B_{2r-1,0}$

which with SIMD shuffling pre-applied becomes:

$Integerify(Bshuf, r) = (Bshuf_{2r-1,13} \ll 32) \vee Bshuf_{2r-1,0}$

(This is the same as implementations of classic scrypt use. Since SIMD shuffling is part of the yescrypt specification for interfacing with pwxform, we also make this detail part of the specification.)

Additionally, for the purpose of interfacing with Salsa20's 32-bit words, pwxform's 64-bit words are assumed to be in little-endian order of their 32-bit halves. This aspect may be reflected in how lo() and hi() as well as the reads from S0 and S1 and the write to $B_{j,k}$ in the pwxform algorithm above are defined, or for better efficiency on 64-bit big-endian architectures (as well as on 32-bit big-endian architectures that have 64-bit loads and stores) optimized implementations may combine the potential 32-bit word swapping along with the SIMD shuffling. These two approaches may be seen implemented in the provided yescrypt-ref.c and yescrypt-opt.c, respectively.

With the optimization mentioned above, the combined SIMD shuffle and potential endianness conversion may be achieved with 8 invocations of COMBINE(out, lo, hi) defined as:

$$Bshuf64_{out} \leftarrow B_{2lo} \vee (B_{2hi+1} \ll 32)$$

```
COMBINE(0, 0, 2)
COMBINE(1, 5, 7)
COMBINE(2, 2, 4)
COMBINE(3, 7, 1)
COMBINE(4, 4, 6)
COMBINE(5, 1, 3)
COMBINE(6, 6, 0)
COMBINE(7, 3, 5)
```

Conversely, the SIMD unshuffle with potential endianness conversion may be achieved with 8 invocations of UNCOMBINE(out, lo, hi) defined as:

$$B_{2out} \leftarrow Bshuf64_{lo} \wedge (2^{32} - 1)$$
$$B_{2out+1} \leftarrow Bshuf64_{hi} \gg 32$$

```
UNCOMBINE(0, 0, 6)
UNCOMBINE(1, 5, 3)
UNCOMBINE(2, 2, 0)
UNCOMBINE(3, 7, 5)
UNCOMBINE(4, 4, 2)
UNCOMBINE(5, 1, 7)
UNCOMBINE(6, 6, 4)
UNCOMBINE(7, 3, 1)
```

When operating on 64-bit integers as above, Integerify() becomes:

$Integerify(Bshuf64, r) = (Bshuf64_{2r-1,6} \gg 32 \ll 32) \vee (Bshuf64_{2r-1,0} \wedge (2^{32} - 1))$

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 is likely to neutralize the effect of 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. 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 the YESCRYPT_RW 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 the code, 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 along with the YESCRYPT_RW flag. 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 the YESCRYPT_RW 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 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.

ASIC and FPGA

yescrypt with the YESCRYPT_RW 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 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, as long as the optional ROM's content is not secret (as it does not need to be for the intended use, where the ROM provides port-hardness).

Thanks

- Colin Percival

- Bill Cox
- Rich Felker
- Anthony Ferrara
- Christian Forler
- Taylor Hornby
- Dmitry Khovratovich

- Samuel Neves
- Marcos Simplicio
- Ken T Takusagawa
- Jakob Wenzel
- Christian Winnerlein

- DARPA Cyber Fast Track