```
at
OK
atdt5551212
CONNECT 2400

Openwall                                                         BSidesLjubljana

                                  present you

                                            __
     __  _____  _________________  ______  / /_
    / / / / _ \/ ___/ ___/ ___/ / / / __ \/ __/
   / /_/ /  __(__  ) /__/ /  / /_/ / /_/ / /_
   \__, /\___/____/\___/_/   \__, / .___/\__/
  /____/                    /____/_/

                        large-scale password hashing

                   by Solar Designer <solar@openwall.com>
                            @solardiz @Openwall
                          http://www.openwall.com

                                March 2017

█
```

Why passwords? Why hashing?

* Passwords remain a convenient and ubiquitous authentication factor

  + "Something you know" in 2FA

* Proper password hashing mitigates the impact of user database leaks

  + Saves a percentage of accounts from compromise until passwords are forcibly changed (as they should be after a known database leak)

  + Mitigates the impact on the users' accounts on other sites, where the same or similar passwords may have been reused

* Password hashing is not a perfect security measure, yet it is a must

  + To make it effective, password policy enforcement is also needed

* A closely related concept is password-based key derivation

* These days, "password" should actually mean "passphrase"

Plaintext password storage (1960s to early 1970s CTSS, TENEX, Unix)

* Setting a new password

```
        password
           |
           v
        password store
```

* CTSS had a password file leak incident through text editor temporary file

* Authenticating with a password

```
   password   password store
           \ /
            v
         compare
   is it timing-safe?
```

* TENEX had a character-by-character timing leak

Password hashing (early 1970s Multics & Unix)

* Setting a new password

```
password
        \
         v
   hash function
         |
         v
password hash store
```

* Authenticating with a password

```
password   password hash store
        \            /
         v          /
   hash function   /
                \ /
                 v
              compare
```

Password hashing (late 1970s Unix)

* Setting a new password

```
                                password   new salt
                           policy check?\ /      |
                                         v       |
                              slow hash function |
                                         |       |
                                         v       v
                                password hash store
```

* Authenticating with a password

```
                              password   password hash store
                                      \ /(salt)    /(hash)
                                       v          /
                            slow hash function   /
                                              \ /
                                               v
                                            compare
```

Password hashing (1990s BSDI, bcrypt, PBKDF2)

* Setting a new password

```
                                password   new salt & cost
                                        \ / (aka setting)
                                         v       |
                      tunably slow hash function |
                                         |       |
                                         v       v
                                password hash store
```

* Authenticating with a password

```
                              password   password hash store
                                      \ /(setting) /(hash)
                                       v          /
                    tunably slow hash function   /
                                              \ /
                                               v
                                            compare
```

yescrypt: large-scale password hashing

Password hashing (2010s scrypt, Argon2, ...)

* Setting a new password

```
                                password   new salt, costs
                                        \ / (aka setting)
                                         v       |
                       memory-hard hash function |
                                         |       |
                                         v       v
                                password hash store
```

* Authenticating with a password

```
                              password   password hash store
                                      \ /(setting) /(hash)
                                       v          /
                     memory-hard hash function   /
                                             \ /
                                              v
                                            compare
```

Password cracking (unsalted, unoptimized)

* For each candidate password
  + For each hash

```
         candidate password   password hash
                           \             /
                            v           /
                      hash function    /
                                   \ /
                                    v
                                 compare
```

Password cracking (unsalted, semi-optimized)

* For each candidate password

```
         candidate password   password hash(es)
                           \             /
                            v           /
                      hash function    /
                                   \ /
                                    v
                     one-to-many compare
```

* We've amortized the cost of hashing, reusing the result of each computation

Password cracking (salted, semi-optimized)

* For each candidate password
  + For each salt

```
         candidate password   password hash(es) that use the current salt
                           \ /(salt)    /(hashes)
                            v          /
                      hash function   /
                                   \ /
                                    v
                     one-to-many compare
             becomes one-to-one when each salt is unique, as they should be
```

* We can no longer amortize the cost of hashing when each salt is unique

  + Caveat: the salt should be unique globally, not just within a database
    (otherwise a multi-target or non-targeted/opportunistic attack on many
    leaked password hash databases would amortize the cost across databases)

yescrypt: large-scale password hashing

Password cracking (salted, fully optimized)

* For each group of candidate passwords (groups of more likely passwords first)
  + For each salt (salts shared by more hashes first)

```
      candidate password(s)   password hash(es) that use the current salt
                           \ /(salt)    /(hashes maybe partially reversed)
                            v          /
         many-to-many hash function   /
         cost-amortizing, optimized\ /
         parallelized               v
                    many-to-many compare
            becomes many-to-one when each salt is unique, as they should be
```

* Cost-amortizing: if a computation doesn't have to be performed per each {password, hash} combination, it is performed less often & result reused

  + "Key setup" and/or "salt setup" may be moved to an outer loop
  + "Finalization" may be reversed at startup & then not performed at all
    + Such as DES final permutation, encoding into an ASCII string, etc.

Password cracking cost reduction

* When we amortize cost, we reduce total cost to achieve the same results

  + We do it e.g. through reducing the total amount of computation

* For well-suited hashing schemes, very little computation can be amortized

  + However, that's not the only way to reduce cost

* Besides computational complexity, the other major metric is space complexity

* Real-world costs may be incurred for hardware, maintenance, energy, etc.

  + These costs are related to both computational and space complexities, as well as to real-world constraints, which may vary by attacker

  + For example, "how many CPUs and how much RAM is occupied for how long, and do we readily have those or do we have to acquire them?"

    + ... and then it might not be CPUs anymore

Password cracking cost reduction through parallel processing

```
* Parallel processing during authentication is limited by the product of:
  + number of concurrent authentication attempts
  + natural parallelism of the password hashing scheme

* Parallel processing potential during password cracking is "unlimited"

* Thus, attack duration can be "arbitrarily" reduced through addition of
  parallel processing elements (CPUs/SIMD, more/larger GPUs/FPGAs/ASICs)

  + along with accordingly more memory
    ... unless the hashing scheme allows for memory cost amortization

    + Most older schemes don't use much memory anyway, so only the cracker's
      memory "overhead" (e.g., hashes) needs to be amortized - and it can be
    + Most modern schemes should avoid this, which they do to varying extent

* Parallel processing doesn't reduce the amount of computation, but
  + it reduces the amount of time for which other resources are held and/or
  + it amortizes their cost (e.g., a CPU alone vs. the CPU+GPUs per chassis)
```

Password cracking cost reduction through time-memory trade-off (TMTO)

```
* It may be possible to compute a function in less time by using more memory
  + e.g., making greater use of table lookups in the computation, or in the
    extreme case replacing the entire computation with a [huge] table lookup

    + Historically, many traditional Unix DES-based crypt(3) crackers used
      larger intermediate lookup tables than defensive implementations did

* Conversely, it may also be possible to compute a function in less memory
  + e.g., by throwing away and recomputing intermediate results when needed,
    which increases the amount of computation (but not necessarily of time)

    + scrypt is deliberately friendly to this trade-off (it lower-bounds the
      time-memory product), and GPU/FPGA/ASIC crackers and miners use it

* The defender's balance for computation (and time) vs. memory usage by a
  password hash function might not match what a given attacker finds optimal

* The attacker may choose to move the balance over the TMTO curve in either
  direction accordingly to this attacker's costs of the different resources
```

Password cracking cost metrics

```
* For a given performance ({password, hash} tests per time, maybe amortized)

  + Hardware: ASIC die area, mm^2
    + for a certain design at a certain clock rate in a certain ASIC process
  + Power, W
    + not a cost per se, but for lengthy attacks translates into energy cost
    + correlates with die area

* For a given attack ("test these candidate passwords against these hashes")

  + Hardware: ASIC die area and time product (area-time, AT), mm^2 * s
  + Energy: power and time product, J = W * s
    + correlates with AT, letting estimate relative costs in AT terms alone

* Hardware and energy may have monetary costs, but not always to the attacker

* Real-world attackers' costs may vary greatly e.g. due to existing hardware,
  which may be cheaper than custom ASIC design yet less optimal for the task
```

yescrypt: large-scale password hashing

Parallelized hash function (originally memoryless)

```
                           candidate passwords
     | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
     v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
     | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
     v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v
  hashes for comparison against those being cracked (for current salt)
```

In this illustration, we're computing 32 hashes in parallel in the same amount of time that a defender using only 1 core (and maybe having only 1 password to authenticate at the moment) would need for 1 hash

Parallelized hash function (amortizable memory-hard)

```
                               candidate passwords
                         | | | | | | | | | | | | | | | |
                         v v v v v v v v v v v v v v v v
    +--------+--------+--------+--------+--------+--------+--------+--------+
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+                                   +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+               memory              +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+                                   +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+--------+--------+--------+--------+
                         | | | | | | | | | | | | | | | |
                         v v v v v v v v v v v v v v v v
      hashes for comparison against those being cracked (for current salt)
```

In this illustration, we're computing 16 hashes in parallel in the same amount of time that a defender using only 1 core (and maybe having only 1 password to authenticate at the moment) would need for 1 hash

yescrypt: large-scale password hashing

Parallelized hash function (parallelizable memory-hard)

```
                           candidate password
                                    |
                                    v
    +--------+--------+--------+--------+--------+--------+--------+--------+
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+                                   +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+               memory              +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+                                   +
    |  core  |  core  |  core  |  core  |                                   |
    +--------+--------+--------+--------+--------+--------+--------+--------+
                                    |
                                    v
        hash for comparison against those being cracked (for current salt)
```

In this illustration, we're computing 1 hash in 1/16 of the amount of time
that a defender using only 1 core and the same amount of memory would spend

Parallelized hash function (sequential memory-hard)

```
                           candidate passwords
                                   | |
                                   v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |                          |  core  |                          |
+--------+                          +--------+                          +
|                                   |                                   |
+               memory              +               memory              +
|                                   |                                   |
+                                   +                                   +
|                                   |                                   |
+--------+--------+--------+--------+--------+--------+--------+--------+
                                   | |
                                   v v
  hashes for comparison against those being cracked (for current salt)
```

In this illustration, we're computing 2 hashes in parallel in the same amount of time that a defender using only 1 core (and maybe having only 1 password to authenticate at the moment) would need for 1 hash, but we need 2x more memory

Segmentation fault (core dumped)

```
                                    +---------+
In these illustrations and further, |  core   | refers to any processing element
                                    +---------+
capable of computing the target hash or an appropriate portion thereof without
having a lot of memory of its own.  We give memory to cores, sometimes sharing
it across several of them.
```

A core may be today's usual CPU core, or it may be a SIMD (single instruction, multiple data) unit (e.g. within a GPU CU or SM), or it may be a SIMD lane (within a CPU core or a GPU SIMD unit), or it may be a pipeline stage (e.g. with different hash computations' instructions interleaved on a superscalar CPU or a GPU), or it may even be a single bit number across N-bit registers in a register file (when we've rotated our problem and are now "bitslicing" it)

Of course, and most importantly, a core may also be a logic circuit in an FPGA or ASIC, but even there by a core we might also be referring e.g. to each pipeline stage, whichever option is relevant or optimal in a given context

Possibilities abound

yescrypt: large-scale password hashing

scrypt memory usage

scrypt spends half its time filling a memory region with password- and salt-
derived blocks and the other half "randomly" indexing and reading those blocks

```
                    sequential write|  random read
                   N|              . . . . . . . . .
                    |            . . . . . . . . . .        100/128
                    |          . . . . . . . . . . .         78%
             memory |        . . . . . . . . . . . .
              usage |      . . . . . . . . . . . . .
                    |    . . . . . . . . . . . . . .
                    |  . . . . . . . . . . . . . . .
                    |. . . . . . . . . . . . . . . .
                   0+--------------------------------
                    0            time             2N
```

Peak memory-time product of defensive use contributes to attack AT cost?
  + No, 1/4 can be amortized across optimally desynchronized instances

3/4 of memory-time product contributes to attack AT cost? Not quite.

scrypt sqrt(N) cores attack

1 core computes blocks yet only stores those with index near a multiple of N/M, then M cores (re)compute the missing blocks from those checkpoints in N/M time

```
        1 core |2 cores|  random read
      N|              . . . . . . . . .
       |            . . . . . . . . . .         88/128
       |          . . . . . . . . . . .         69%
memory |        . . . . . . . . . . . .
 usage |              . . . . . . . . .
       |            . . . . . . . . . .
       |          . . . . . . . . . . .
       |. . . . . . . . . . . . . . . .
      0+-------------------------------
       0            time             2N
```

Asymptotically, AT cost of the memory filling phase is negligible
  + In the real world, each core would also need a write port to memory

1/2 of memory-time product contributes to attack AT cost? Not quite.

scrypt TMTO attack

We can avoid storing some of the blocks throughout the entire computation, and recompute the missing blocks from the preceding checkpoints if and when needed

```
        1 core |2 cores|random read & recompute
      N|
       |              . . . . . . . . . . . . . 62/128
       |                                        48%
memory |        . . . . . . . . . . . . . . . .
 usage |
       |              . . . . . . . . . . . . .
       |
       |. . . . . . . . . . . . . . . . . . . .
      0+---------------------------------------
       0            time             2N   ~2.5N
```

scrypt TMTO attack

We can avoid storing some of the blocks throughout the entire computation, and recompute the missing blocks from the preceding checkpoints if and when needed

```
                     1 core |2 cores|random read & recompute
                 N/2|              . . . . . . . . . . . . .
             memory |        . . . . . . . . . . . . . . . . 62/128
              usage |              . . . . . . . . . . . . . 48%
                    |. . . . . . . . . . . . . . . . . . . .
                   0+---------------------------------------
                    0            time             2N   ~2.5N
```

* Asymptotically, only 1/4 of the memory-time product of a straightforward (usually optimal) defensive implementation contributes to attack AT cost

* It's 1/4 even without the sqrt(N) cores attack as the total running time becomes so high that the memory filling time is negligible in comparison

* Not an oversight: attack cost estimates in scrypt paper include this 1/4

yescrypt memory usage

yescrypt combines sequential writes and random reads (of the blocks already written) until its memory is full and then proceeds with random read-writes

```
         seq w & rnd r |  random r/w
      N|              . . . . . . . . .
       |            . . . . . . . . . .       100/128
       |          . . . . . . . . . . .        78%
memory |        . . . . . . . . . . . .
 usage |      . . . . . . . . . . . . .
       |    . . . . . . . . . . . . . .
       |  . . . . . . . . . . . . . . .
       |. . . . . . . . . . . . . . . .
      0+-------------------------------
       0            time            2N
```

yescrypt sliding window

During the memory filling phase, random reads occur from a sliding window.
Like N, the size of the sliding window is a power of two (2, 4, ..., N/2).

```
                      seq w & rnd r |  random r/w
                   N|              . . . . . . . . .
                    |    29/36   . | . . . . . . . .        100/128
                    |    81%   . | | . . . . . . . .         78%
             memory |        . | | | . . . . . . . .
              usage |      . | | | | . . . . . . . .
                    |    . | | | | . . . . . . . . .
                    |  . | | | | . . . . . . . . . .
                    |. . | . | . . . . . . . . . . .
                   0+-------------------------------
                    0            time             2N
```

Compared to alternatives such as modulo division to obtain an index over the
full range, the sliding window provides greater assurance regarding a smaller
AT cost (1/3*N^2 vs. the potential of 1/2*N^2) and doesn't require an extra
potentially timing-unsafe operation (bitwise AND and ADD vs. modulo division)

yescrypt sliding window

During the memory filling phase, random reads occur from a sliding window.
Like N, the size of the sliding window is a power of two (2, 4, ..., N/2).

```
         seq w & rnd r |  random r/w
      N|              . . . . . . . . .
       |    29/36   . | . . . . . . . .        100/128
       |    81%   . | | . . . . . . . .         78%
memory |        . | | | . . . . . . . .
 usage |      . | | | | . . . . . . . .
       |    . | | | | . . . . . . . . .
       |  . | | | | . . . . . . . . . .
       |. . | . | . . . . . . . . . . .
      0+-------------------------------
       0            time             2N
```

With the divisor starting small and increasing towards N, modulo division would be heavily biased towards smaller block numbers. This bias could be compensated for with more math, but sliding window does the trick on its own and it indexes more distinct blocks (~57% vs. 50% by end of memory filling).

yescrypt random read-writes

Once the memory is full, yescrypt proceeds with random read-writes - that is, it not only uses each random block for computation but also updates the block

```
                      seq w & rnd r |  random r/w
                   N|              . | | | | | | | |
                    |    29/36   . | | | | | | | | |        100/128
                    |    81%   . | | | | | | | | | |         78%
             memory |        . | | | | | | | | | | |
              usage |      . | | | | | | | | | | | |
                    |    . | | | | . | | | | | | | |
                    |  . | | | | . . | | | | | | | |
                    |. . | . | . . . | | | | | | | |
                   0+-------------------------------
                    0            time             2N
```

This increases the number and variety of prior blocks that the updated blocks depend on, which in turn increases the amount of recomputation and each block's recomputation tree depth (and thus time to reach leaf nodes) in a TMTO attack

yescrypt swap in/out attack

It may be practical to swap out and then back in the blocks that fall outside of the sliding window, leaving us with:

```
                      seq w & rnd r |  random r/w
                   N|              . | | | | | | | |
                    |    29/36   . | | | | | | | | |         93/128
                    |    81%   . | | | | | | | | | |         73%
             memory |        . | | | | | | | | | | |
              usage |      . | | | | | | | | | | | |
                    |    . | | | |   | | | | | | | |
                    |  . | | | |     | | | | | | | |
                    |. . |   |       | | | | | | | |
                   0+-------------------------------
                    0            time             2N
```

# Asymptotically, 2/3 of the peak memory-time product goes to attack AT cost

# That's much better than scrypt's 1/4. Can we do better yet? In a sense.

yescrypt optimal running time

We should actually want to maximize attack cost achievable in a given defensive running time budget. Turns out it's no longer optimal to go to 2N.

```
                      seq w & rnd r |  random r/w
                   N|              . | | |
                    |    29/36   . | | | |                   53/88
                    |    81%   . | | | | |                   60%
             memory |        . | | | | | |
              usage |      . | | | | | | |
                    |    . | | | |   | | |
                    |  . | | | |     | | |
                    |. . |   |       | 100%  96%  89%
                   0+------------------t=0----1----2
                    0            time  4N/3       2N
```

Due to our improvements to the memory filling phase (which in scrypt didn't contribute to asymptotic AT cost, but in yescrypt it does), it's now optimal to end our defensive yescrypt runs after indexing random blocks only N/3 times in the read-writes phase. This enables the defender to use more memory.

yescrypt higher memory usage in same time

If we "reduce" our running time from 2N to 4N/3, we can increase N*r by 50%

```
                     seq write & random read|rnd r/w
                 N*r|                      . | | | |
              block^|                    . | | | | |
               size |    60/78         . | | | | | |        108/192
                    |    77%         . | | | | | | |         56%
             memory |              . | | | | | | | |
              usage |            . | | | | | | | | |        108/128
                    |          . | | | | | | | | | |         84%
                    |        . | | | | | |   | | | |
                    |      . | | | | | |     | | | |
                    |    . | | | | | |       | | | |
                    |  . . | |   | |         | | | |
                    |. . . |     |           | | | |
                   0+----------------------------t=0
                    0            time           4N/3
```

* Asymptotically, 1/2 of new or 3/4 of old N*r's memory-time goes to AT cost

yescrypt higher memory usage in same time?

yescrypt's 4N/3 block operations at 50% higher N*r process the same number of bytes as scrypt's 2N block operations do. Is it the same amount of real time?

* yescrypt uses twice more bandwidth: each sequential write is accompanied with a random read, and each random read is accompanied with a rewrite (cache-friendlier than a random write unrelated to a prior read would be)

  + If we can fit this in the same defensive running time, that's great - a way to further increase cost of attacks with other/specialized hardware

  + Can we? Depends on concurrency, bandwidth saturation, full/half duplex.

* There are also differences in how we process the contents of each block (naturally, we don't only write and read - we also compute)

* In some benchmarks, yescrypt at 4N/3 is as fast as scrypt at 2N for same N, meaning we can't use more memory and so we "merely" double the AT cost

* In others, we can and so we triple the AT cost vs. scrypt's for same time

yescrypt TMTO attacks?

So far, we assumed no AT cost reducing TMTO attacks on yescrypt are possible besides swapping in/out the blocks that fall outside of the sliding window. Is this so? Probably yes.

* A further TMTO attack would involve not storing and then recomputing some intermediate data (the pieces of data not stored would need to be larger than the stored information on where to find the inputs to recompute those pieces)

* Two metrics are increases in computation and time (recomputation tree depth)

* Simulation shows disproportionately high increases in amount of computation when the amount of memory is decreased (~3x for 1/2, ~1000x for 1/4 at t=0, and the increases should only be steeper at higher t)

* Until a further change, a massively parallel sub-block level TMTO attack would make the increases of time too small at moderate decreases of memory

* The change (in 0.8.1, PHC v2, October 2015) addressed sub-block level attacks and increased the amount of recomputation further (beyond the figures above)

Why sub-blocks?

In (ye)scrypt, each block consists of 2r sub-blocks of 64 bytes each.
For example, at r=8 each 1 KiB block consists of 16 sub-blocks.

Here's why:

* We need to make the blocks larger to amortize the cost of TLB misses and memory latency in defensive implementations

* We need sub-blocks to make the blocks larger than the underlying primitive's input and output sizes (which are 64 bytes for scrypt's Salsa20/8)

* In yescrypt, we also use this opportunity to amortize the cost of mixing of the SIMD lanes (more on this later), which is only done once per full block

Sub-blocks

In (ye)scrypt, each block consists of 2r sub-blocks of 64 bytes each.
For example, at r=8 each 1 KiB block consists of 16 sub-blocks.

```
         0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         1 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         3 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         5 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
         7 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
       N=8
```

In this illustration, each character represents 16 bytes of data, and we set the characters to hexadecimal zero-based numbers of the sub-blocks

Suppose each sub-block in a newly computed block (to be (re)written) depended on the same-numbered sub-block(s) in its input block(s). Then the hexadecimal digits above would also represent the data dependencies for (re)computation.

Sub-block level TMTO attack setup

Suppose that besides same-numbered sub-block(s), there are also early data dependencies on the last sub-blocks. Don't store the rest in some blocks.

```
        0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        1                                                             FFFF
        2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        3                                                             FFFF
        4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        5                                                             FFFF
        6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        7
      N=8
```

Suppose we're computing block 7, which depends on block 5, which in turn depends on block 3, and which in turn depends on block 1 (as well as possibly on some blocks that we did store in full)

The mostly-missing blocks also depend on their immediately preceding blocks, which we did store in full

Sub-block level TMTO attack progress

We start by recomputing first sub-block of block 1, which we can do right away since we have the preceding block's first and last sub-blocks

```
        0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        1 --->                                                        FFFF
        2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        3                                                             FFFF
        4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        5                                                             FFFF
        6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        7
      time   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

Suppose we're computing block 7, which depends on block 5, which in turn depends on block 3, and which in turn depends on block 1 (as well as possibly on some blocks that we did store in full)

The mostly-missing blocks also depend on their immediately preceding blocks, which we did store in full

Sub-block level TMTO attack progress

Then on 2 cores we simultaneously recompute second sub-block of block 1 and first sub-block of block 3, using the recomputed first sub-block of block 1

```
  0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
  1 ------->                                                    FFFF
  2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
  3 --->                                                        FFFF
  4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
  5                                                             FFFF
  6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
  7
time   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

Suppose we're computing block 7, which depends on block 5, which in turn depends on block 3, and which in turn depends on block 1 (as well as possibly on some blocks that we did store in full)

The mostly-missing blocks also depend on their immediately preceding blocks, which we did store in full

Sub-block level TMTO attack progress

Then on 3 cores we simultaneously recompute third sub-block of block 1, second sub-block of block 3, and first sub-block of block 5

```
        0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        1     ------->                                                FFFF
        2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        3 ------->                                                    FFFF
        4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        5 --->                                                        FFFF
        6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        7
      time   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

We can start throwing away the already used recomputed sub-blocks, as shown for first sub-block of block 1 above

Sub-block level TMTO attack progress

Finally, we proceed on 4 cores to recompute the remaining sub-blocks of blocks 1, 3, 5, and our target block 7

```
        0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        1         ------->                                            FFFF
        2 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        3     ------->                                                FFFF
        4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        5 ------->                                                    FFFF
        6 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
        7 --->
      time   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

With this approach, we'd recompute our missing 60 sub-blocks in 18 sub-blocks' worth of time. Our amount of computation is still that for 60 sub-blocks, but our time spent (latency) is lower (and thus our ASIC or whatever is holding up its precious memory for a shorter period, advancing to next candidate sooner)

As the scrypt paper casually says, "the computations would neatly `pipeline'"

scrypt sub-block shuffling

To mitigate pipelined recomputation in sub-block level TMTO attacks, scrypt shuffles the sub-blocks in a pre-determined manner on each block computation

```
          0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
          1 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          2 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          3 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          4 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          5 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          6 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
          7 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
        N=8
```

In this illustration, we set the characters to sub-block numbers for block 0, and to each block's preceding block's sub-block numbers representing the data dependencies for subsequent blocks

This no longer pipelines so neatly, but recomputation time can still be lower than sequential's, especially if we store occasional checkpoint sub-blocks

scrypt sub-block shuffling

To mitigate pipelined recomputation in sub-block level TMTO attacks, scrypt shuffles the sub-blocks in a pre-determined manner on each block computation

```
       0 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
       1 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
       2 000044448888CCCC111155559999DDDD22226666AAAAEEEE33337777BBBBFFFF
       3 000088881111999922222AAAA3333BBBB4444CCCC5555DDDD6666EEEE7777FFFF
       4 0000111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFF
       5 00002222444466668888AAAACCCCEEEE11113333555577779999BBBBDDDDFFFF
       6 000044448888CCCC111155559999DDDD22226666AAAAEEEE33337777BBBBFFFF
       7 00008888111199992222AAAA3333BBBB4444CCCC5555DDDD6666EEEE7777FFFF
     N=8
```

In this illustration, we set the characters to sub-block numbers for block 0, and to ultimate data dependencies (going all the way back to block 0) for subsequent blocks. It's a different way to look at the same shuffling scheme.

This no longer pipelines so neatly, but recomputation time can still be lower than sequential's, especially if we store occasional checkpoint sub-blocks

Local memory

Not all memory is the same. While scrypt targets a typical computer's RAM with ~1 KiB lookups from a multi-megabyte region, bcrypt targets L1 cache with rapid 4 byte lookups from a 4 KiB region. As it happens, bcrypt is relatively slow to crack on GPUs - in some cases, matching GPU attack / CPU defense speed ratio of scrypt at multiple megabytes. Why is that? Local memory exhaustion.

* GPUs are designed for much higher concurrency than CPUs, but have relatively little local memory (even if more than a CPU does in absolute terms)

  + As a result, there's less local memory per hash being computed (work item), leaving the GPU's computing units under-utilized

  + Design for best performance at higher concurrency means higher latencies

* Going out to off-chip global memory for 4 byte quantities is inefficient

  + Memory buses tend to be much wider than that

  + Memory fetches tend to be of entire cache lines, which are also much wider

yescrypt local memory usage

* While scrypt does benefit from some cache or local memory for the block it's currently working on, it does not specifically target a cache or local memory

* yescrypt does, with a Blowfish-like component in its sub-block processing, in place of scrypt's use of Salsa20/8

* The aim is for yescrypt to be on par with bcrypt at GPU attack resistance even at unusually low memory settings (kilobytes)

* This might achieve little against FPGAs and ASICs, but even then GPU attacks are also relevant because many attackers readily have GPUs

* That said, it also helps against FPGAs and ASICs in terms of computation latency hardening (more on this later)

yescrypt sub-block processing

* yescrypt uses a custom primitive called pwxform to process sub-blocks

* After the last sub-block of a block, pwxform's lanes are mixed with Salsa20/2

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. This lets it be adapted to a wide variety of current and future scalar and SIMD instruction sets and varying instruction level parallelism, which defensive uses of yescrypt would take advantage of.

As currently recommended, yescrypt's pwxform is instantiated for 2-wide simple SIMD lanes, 4-wide gather SIMD vector, 12 KiB S-boxes (3x 4 KiB), and 6 rounds

S-boxes compare favorably to bcrypt's (8 KiB active for random lookups at any given time vs. bcrypt's 4 KiB). Parallelism is twice higher (8 independent S-box lookups per round vs. bcrypt's 4). Size of individual lookups is much higher (16 bytes vs. bcrypt's 4 bytes), but that's actually an advantage if it doesn't slow them down in defensive implementations. Overall, it's comparable.

yescrypt pwxform (one round, all simple SIMD lanes)

```
 +---+---+---+128  +---+---+---+128  +---+---+---+128  +---+---+---+128
 |32 |32 |32 |32   |32 |32 |32 |32   |32 |32 |32 |32   |32 |32 |32 |32
 v   v   v   v     v   v   v   v     v   v   v   v     v   v   v   v
  MUL    |MUL|      MUL    |MUL|      MUL    |MUL|      MUL    |MUL|
   |64   v | v       |64   v | v       |64   v | v       |64   v | v
   |     S1| S0      |     S1| S0      |     S1| S0      |     S1| S0
   |  128| | |128    |  128| | |128    |  128| | |128    |  128| | |128
 +-------+ | |     +-------+ | |     +-------+ | |     +-------+ | |
 | |       | |     | |       | |     | |       | |     | |       | |
 | |     +-+ |     | |     +-+ |     | |     +-+ |     | |     +-+ |
 | |     |   |     | |     |   |     | |     |   |     | |     |   |
 | |   +-----+     | |   +-----+     | |   +-----+     | |   +-----+
 | | Hi| |   |Lo   | | Hi| |   |Lo   | | Hi| |   |Lo   | | Hi| |   |Lo
 | | 64| |64 |64   | | 64| |64 |64   | | 64| |64 |64   | | 64| |64 |64
 | v   v v   v     | v   v v   v     | v   v v   v     | v   v v   v
 |  ADD   ADD      |  ADD   ADD      |  ADD   ADD      |  ADD   ADD
 v   |64   |64     v   |64   |64     v   |64   |64     v   |64   |64
XOR<-+-----+      XOR<-+-----+      XOR<-+-----+      XOR<-+-----+
 |128              |128              |128              |128
```

yescrypt: large-scale password hashing

yescrypt pwxform (one round, one simple SIMD lane)

```
 +---+---+---+128
 |32 |32 |32 |32
 v   v   v   v
  MUL    |MUL|
   |64   v | v
   |     S1| S0
   |  128| | |128
 +--------+ | |
 | |       | |
 | |     +-+ |
 | |     |   |
 | |   +-----+
 | | Hi| |   |Lo
 | | 64| |64 |64
 | v   v v   v
 |  ADD   ADD
 v   |64   |64
XOR<-+-----+
 |128
```

This might look complex, but it fits typical SIMD instruction sets like SSE2 well

32x32->64 integer multiplication and the S-box lookups provide computation latency hardening, currently ensuring attack latency of around 1 ns

```
uint64_t x;
uint32_t lo = x = _mm_cvtsi128_si64(X) & Smask2reg;
uint32_t hi = x >> 32;
X = _mm_mul_epu32(_mm_shuffle_epi32(X, 0xb1), X);
X = _mm_add_epi64(X, *(__m128i *)(S0 + lo));
X = _mm_xor_si128(X, *(__m128i *)(S1 + hi));
```

The output of each round is fed into next round. Except on first and last round, it is also written into S2, the third S-box. Pointers to the S-boxes are frequently rotated.

yescrypt pwxform (one round, one simple SIMD lane)

```
 +---+---+---+128
 |32 |32 |32 |32
 v   v   v   v
  MUL    |MUL|
   |64   v | v
   |     S1| S0
   |  128| | |128
 +--------+ | |
 | |       | |
 | |     +-+ |
 | |     |   |
 | |   +-----+
 | | Hi| |   |Lo
 | | 64| |64 |64
 | v   v v   v
 |  ADD   ADD
 v   |64   |64
XOR<-+-----+
 |128
```

In fact, it's just 8 instructions on x86-64
(SSE2 is a genuine part of x86-64 architecture)

32x32->64 integer multiplication and the S-box
lookups provide computation latency hardening

```
movd X, %rax                Sorry Intel, pardon my AT&T
pshufd $0xb1, X, H
andq Smask2, %rax
pmuludq H, X                UMLAL [VMLAL] on ARM [NEON]
movl %eax, %ecx
shrq $0x20, %rax
paddq (S0,%rcx), X
pxor (S1,%rax), X
```

The output of each round is fed into next round.
Except on first and last round, it is also written
into S2, the third S-box. Pointers to the S-boxes
are frequently rotated.

yescrypt pwxform vs. sub-block level TMTO attacks

yescrypt does not implement scrypt's sub-block shuffling, but its pwxform is even more effective at making sub-block level TMTO attacks impractical

* With the current instantiation of pwxform (6 rounds), 4 times as much data is written into the S-boxes as is written into the output block

  + Those extra writes are almost free on typical CPUs since L1 data caches have separate read and write ports anyway (the write ports would be idle)

* With the weakest possible instantiation of pwxform (3 rounds), as much data is written into the S-boxes as is written into the output block

* In a TMTO attack, not only some checkpoint (sub-)blocks, but also contents of the S-boxes at some points would need to be stored

  + Storing the S-box contents defeats the purpose of the attack, since it's at least as much new data to store as there would be in the (sub-)blocks

* With its S-box lookups and rewrites, pwxform is similar to yescrypt itself

yescrypt computation latency hardening

* scrypt's Salsa20/8 is optimally computable in a lot fewer clock cycles (and likely in less time) on ASIC than on CPU

* yescrypt's 32x32 to 64-bit integer multiplication and 4 KiB S-box lookups are unlikely to see as much speedup

* The latency of each pwxform round is lower-bound by the maximum of the two latencies of integer multiplication and S-box lookups

  + We're performing these operations in parallel

* An attack trying to improve the time factor in AT by simply computing things quicker would need to improve both of these latencies (one is not enough)

* On modern fast CPUs, either of these operations has a few cycles of latency (commonly 3 to 5 for MUL, 4 for L1 cache load), which at 4 GHz translates to about 1 ns

* Unfortunately, we also incur some "overhead" (simpler 1-cycle operations)

yescrypt: large-scale password hashing

scrypt defensive thread-level parallelism

scrypt's thread-level parallelism parameter works as a multiplier of scrypt instances discussed so far. In software, it's optionally parallel outer loop.

```
          password (or passphrase), salt, parameters (including e.g. p=2)
                                             |
                                             v
     +---------+---------+---------+---------+---------+---------+---------+---------+
     |  core   |                             |  core   |                             |
     +---------+                             +---------+                             +
     |                                       |                                       |
     +                memory                 +                memory                 +
     |                                       |                                       |
     +                                       +                                       +
     |                                       |                                       |
     +---------+---------+---------+---------+---------+---------+---------+---------+
                                             |
                                             v
                                 hash (or derived key)
```

scrypt defensive thread-level parallelism

scrypt's thread-level parallelism parameter works as a multiplier of scrypt instances discussed so far. In software, it's optionally parallel outer loop.

p=2 (parallelized)

```
      sequential write|  random read         sequential write|  random read
     N|              . . . . . . . . .      N|              . . . . . . . . .
    m |            . . . . . . . . . .     m |            . . . . . . . . . .
    e |          . . . . . . . . . . .     e |          . . . . . . . . . . .
    m |        . . . . . . . . . . . .     m |        . . . . . . . . . . . .
    o |      . . . . . . . . . . . . .     o |      . . . . . . . . . . . . .
    r |    . . . . . . . . . . . . . .     r |    . . . . . . . . . . . . . .
    y |  . . . . . . . . . . . . . . .     y |  . . . . . . . . . . . . . . .
      |. . . . . . . . . . . . . . . .       |. . . . . . . . . . . . . . . .
     0+-------------------------------      0+-------------------------------
      0            time             2N       0            time             2N
```

* When the loop is parallelized, the memory regions have to be separate, so the
  memory usage is multiplied by p

scrypt running time tuning

scrypt's thread-level parallelism parameter works as a multiplier of scrypt instances discussed so far. In software, it's optionally parallel outer loop.

p=2 (not parallelized)

```
         sequential write|  random read |sequential write|  random read
        N|              . . . . . . . . .               . . . . . . . . .
       m |            . . . . . . . . . .             . . . . . . . . . .
       e |          . . . . . . . . . . .           . . . . . . . . . . .
       m |        . . . . . . . . . . . .         . . . . . . . . . . . .
       o |      . . . . . . . . . . . . .       . . . . . . . . . . . . .
       r |    . . . . . . . . . . . . . .     . . . . . . . . . . . . . .
       y |  . . . . . . . . . . . . . . .   . . . . . . . . . . . . . . .
         |. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
        0+----------------------------------------------------------------
         0              N              2N              3N     time     4N
```

* When the loop is not parallelized, each iteration can reuse the same memory
  region, so higher p merely increases running time without using more memory

scrypt running time tuning

When the loop is not parallelized, p is a way to increase running time without using more memory. Elegant (not needing a separate parameter) or suboptimal?

p=2 (not parallelized)

```
         sequential write|  random read |sequential write|  random read
        N|              . . . . . . . . .               . . . . . . . . .
       m |            . . . . . . . . . .             . . . . . . . . . .
       e |          . . . . . . . . . . .           . . . . . . . . . . .
       m |        . . . . . . . . . . . .         . . . . . . . . . . . .
       o |      . . . . . . . . . . . . .       . . . . . . . . . . . . .
       r |    . . . . . . . . . . . . . .     . . . . . . . . . . . . . .
       y |  . . . . . . . . . . . . . . .   . . . . . . . . . . . . . . .
         |. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
        0+---------------------------------------------------------------
         0              N              2N              3N     time     4N
```

* 1/4 of memory-time product can be amortized across optimally desynchronized instances (not considering other attacks). Can we do better?

yescrypt running time tuning

scrypt's p allowing for running time tuning (albeit after compile-time(?) choice not to parallelize) may be elegant, but it's also suboptimal

p=1, t=4

```
           seq w & rnd r |                  random r/w
        N|              . . . . . . . . . . . . . . . . . . . . . . . . .
       m |            . . . . . . . . . . . . . . . . . . . . . . . . . .
       e |          . . . . . . . . . . . . . . . . . . . . . . . . . . .
       m |        . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       o |      . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       r |    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       y |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
         |. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
        0+------------------t=0----1----2---------------3---------------4
         0              N   4N/3       2N              3N     time     4N
```

* 1/8 of memory-time product can be amortized across optimally desynchronized instances (not considering other attacks), an improvement over scrypt's 1/4

yescrypt running time tuning

yescrypt's t parameter allows for (fine-)tuning the running time separately from memory usage and parallelism. t=1 and t=2 are good for fine-tuning.

p=1, t=4

```
           seq w & rnd r |                  random r/w
        N|              . . . . . . . . . . . . . . . . . . . . . . . . .
       m |            . . . . . . . . . . . . . . . . . . . . . . . . . .
       e |          . . . . . . . . . . . . . . . . . . . . . . . . . . .
       m |        . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       o |      . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       r |    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       y |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
         |. . . . . . . . . 100%. 96% .89%. . . . . .69%. . . . . .56%
        0+------------------t=0----1----2---------------3---------------4
         0              N   4N/3       2N              3N     time     4N
```

* Normalized AT cost relative to what's optimal in same defensive running time decreases, but that's OK if we couldn't afford more memory (use case for t)

YESCRYPT_RW vs. YESCRYPT_WORM

* yescrypt deviates from scrypt heavily, but sometimes it may be preferable to tweak scrypt just a little bit

* YESCRYPT_RW is full yescrypt, and YESCRYPT_WORM is a minimal scrypt tweak

  + RW stands for read-write, WORM stands for write once, read many times

	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%

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
          password (or passphrase), salt, parameters (including e.g. p=2)
                                             |
                                             v
                         +---------+---------+---------+---------+
                         |  core   |         |  core   |         |
                         +---------+         +---------+         +
                         |                   |                   |
                         +      memory       |      memory       +
                         |       r/w         |       r/w         |
                         +                   |                   +
                         |                   |                   |
                         +---------+---------+---------+---------+
                                             |
                                             v
                                  hash (or derived key)
```

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
password (or passphrase), salt, parameters (including e.g. p=2)
                               |
                               v
             +--------+--------+--------+--------+
             |  core  |        |  core  |        |
             +--------+        +--------+        +
             |                                   |
             +               memory              +
             |                r/o                |
             +                                   +
             |                                   |
             +--------+--------+--------+--------+
                               |
                               v
                     hash (or derived key)
```

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
           seq w|rnd|                  p=2
           rnd r|r/w|                       random read
        N|      . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       m |    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       e |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       m |. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       o | - - -.-.-. . . . . . . . . . . . . . . . . . . . . . . . . . .
       r |    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       y |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
         |. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
        0+---------t=0---------------------------------------------------
         0     N/2 2N/3 N              2N              3N     time     4N
```

* With per-thread memories, accesses are read/write

* Once the memories are combined, accesses have to become read-only

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
           seq w|rnd|rnd               p=2
           rnd r|r/w|r/o
        N|      . . . .     Why divider?  To keep N a power of 2 when the
       m |    . . . . .     memories are combined, given arbitrary p.
       e |  . . . . . .     To use more memory and time, increase N*r.
       m |. . . . . . .
       o | - - -.-.-. .
       r |    . . . . .
       y |  . . . . . .
         |. . . . . . .
        0+---------t=0---------------------------------------------------
         0     N/2 2N/3 N              2N              3N     time     4N
```

* With per-thread memories, accesses are read/write

* Once the memories are combined, accesses have to become read-only

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
           seq w|rnd|rnd               p=2
           rnd r|r/w|r/o
        N|      . . . .     Why divider?  To keep N a power of 2 when the
       m |    . . . . .     memories are combined, given arbitrary p.
       e |  . . . . . .     To use more memory and time, increase N*r.
       m |. . . . . . .
       o | - - -.-.-. .     Why brief random read-writes?  So that there's
       r |    . . . . .     clearly no attack cost decrease when N and p
       y |  . . . . . .     are increased by the same factor.  Together,
         |. . . . . . .     the p threads do just as many read-writes.
        0+---------t=0---------------------------------------------------
         0     N/2 2N/3 N              2N              3N     time     4N
```

* With per-thread memories, accesses are read/write

* Once the memories are combined, accesses have to become read-only

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
           seq w|rnd|rnd                p=2
           rnd r|r/w|r/o
        N|      . . . .     Why divider?  To keep N a power of 2 when the
       m |    . . . . .     memories are combined, given arbitrary p.
       e |  . . . . . .     To use more memory and time, increase N*r.
       m |. . . . . . .
       o | - - -.-.-. .     Why brief random read-writes?  So that there's
       r |    . . . . .     clearly no attack cost decrease when N and p
       y |  . . . . . .     are increased by the same factor.  Together,
         |. . . . . . .     the p threads do just as many read-writes.
        0+---------t=0----------------------------------------------------
         0     N/2 2N/3 N              2N              3N     time      4N
                    ^ ^
                    | |
       Only two thread sync points: when combining the memory regions and
       at the very end (scrypt has only one sync point at the very end)
```

yescrypt defensive thread-level parallelism

yescrypt's thread-level parallelism parameter works as a divider, initially allocating portions of the memory region to threads and then combining them

```
           seq w|rnd|rnd               p=2
           rnd r|r/w|r/o
        N|      . . . .     The attacker has to provide the full amount of
       m |    . . . . .     memory (or at least swap space) too, although
       e |  . . . . . . .     the dependency on it being RAM is greatest at
       m |. . . . . . . .     higher t.  This limits flexibility of attacks.
       o | - - -.-.-. .
       r |    . . . . .
       y |  . . . . . . .
         |. . . . . . . .
        0+---------t=0---------------------------------------------------
         0     N/2 2N/3 N              2N              3N     time     4N
          ^^^^^^^^^^ ^^^
          |            + needs full memory (or swap in) even if sequential
       can be
       sequential in less RAM, but all data has to be stored (swapped out)
```

scrypt for mass user authentication

When trying to use scrypt for mass user authentication e.g. at an "Internet company", the throughput and latency constraints are commonly such that its memory usage has to be low (e.g. 2 MiB to achieve 7500 requests per second on our reference 2x E5-2670 with 8x DDR3-1600 running 32 hardware threads).

This works, but it isn't ideal. Facebook uses scrypt at 16 MiB off-loaded to frontend nodes (whatever that means). That's great, but not everyone will.

* While low running time implies correspondingly low attack cost, low memory usage exacerbates that since attacks' AT cost goes as product of these two

* At low memory usage like 2 MiB, scrypt might be on par with or weaker than bcrypt in terms of GPU attacks (even if much stronger against ASIC attacks)

  + We're already dealing with that in yescrypt by pwxform's S-box lookups

* scrypt is efficient to compute on smaller systems including botnet nodes

  + Ideally, we'd have the hashes benefit from server hardware more

yescrypt for mass user authentication

What if we pre-fill memory once and reuse it across hash computations? Then our memory filling time isn't limited by the time allotted to compute a hash, so we can use a lot more memory, but our hash function is no longer sequential memory-hard - rather, it is amortizable memory-hard. Since the memory can be shared across cores, its cost can be amortized and thus it no longer directly contributes to AT cost of attacks.

Defensive implementations may benefit from this memory sharing too, but not being sequential memory-hard and thereby allowing for greater cost amortization in attacks than we use in defense is a drawback. Can we do better? Yes.

We can keep the sequential memory-hard hash function, letting it use whatever memory it can fill in time, and enhance it to also use the larger pre-filled shared memory. It is still possible to amortize the shared memory's cost, but not the costs of private memories of individual hash computations.

In yescrypt, we call the previously discussed private memories "RAM" and the pre-filled shared memory "ROM"

yescrypt ROM

If the cost of ROM can be amortized, does it help at all? Yes, and here's how:

* Sharing a memory across cores involves costs: memory ports and interconnect and bandwidth usage (which translates into energy consumption)

  + We jokingly say that yescrypt with a ROM is "ROM-port-hard"

* For a lot of cores sharing a ROM, the cost of such sharing is substantial and potentially prohibitive (at some point it's cheaper to duplicate the ROM)

* For "few" cores, the ROM is larger than all RAMs combined (e.g., 112 GiB ROM is 56x larger than 1000 cores' 2 MiB RAMs combined)

  + This probably makes the ROM costlier than the RAMs as well, although that depends on the respective memory types and on how cost scales with size

* The ROM ideally should not fit into RAM of common attack hardware, such as botnet nodes and GPUs, making computation of yescrypt hashes with the ROM inefficient on those devices and letting us benefit from server hardware more

yescrypt ROM

Here is an illustration of relative sizes (perhaps on a logarithmic scale) of cores, RAMs, and ROM, also showing some of the required interconnect

```
    +--------+--------+--------+--------+--------+--------+--------+--------+
    |        |  core  |                                                     |
    +  RAM   +H------H+                                                     +
    |    port=#      #=port                                                 |
    +--------+        |                         ROM                         +
    |    port=#      #=port                                                 |
    +  RAM   +H------H+                                                     +
    |        |  core  |                                                     |
    +--------+--------+--------+--------+--------+--------+--------+--------+
```

For mass user authentication, the defensive parallelism may currently be assumed to come from concurrent authentication requests, with no need to use yescrypt's p parameter (just keep it at p=1). There's also no need to synchronize the hash computations (except maybe to reduce L3 cache thrashing). When there are not enough concurrent authentication requests, this simply means the server isn't loaded to its full capacity at the moment - that's OK.

yescrypt ROM initialization

We could use e.g. data from /dev/urandom stored on SSD, which would be OK, but yescrypt does offer its own way to initialize the ROM from a seed quickly

```
             +--------+--------+--------+--------+--------+--------+--------+
             |  core  |                                                     |
             +-------H+                                                     +
                     #=port                  will be                        |
                      |                        ROM                          +
                     #=port                                                 |
             +-------H+                                                     +
             |  core  |                                                     |
             +--------+--------+--------+--------+--------+--------+--------+
```

We reuse yescrypt itself, along with its thread-level parallelism support, to have the threads fill what they "think" are their private memories, but is actually our future ROM. In this special mode, yescrypt tells itself to expand the random read-writes phase to cover the running time that would have been spent on the unneeded random reads phase (which wouldn't have affected the memory region's contents). There are no RAMs yet during ROM initialization.

yescrypt ROM initialization

First, each thread fills its "RAM" (actually its portion of the future ROM) with seed-derived random-looking data. To illustrate, we use thread numbers.

```
             +--------+--------+--------+--------+--------+--------+--------+
             |  core  |11111111111111111111111111111111111111111111111111111|
             +-------H+11111111111111111111111111111111111111111111111111111+
                     #=11111111111111111111111111111111111111111111111111111|
                      |111111111111111111111111111222222222222222222222222222+
                     #=22222222222222222222222222222222222222222222222222222|
             +-------H+22222222222222222222222222222222222222222222222222222+
             |  core  |22222222222222222222222222222222222222222222222222222|
             +--------+--------+--------+--------+--------+--------+--------+
```

We reuse yescrypt itself, along with its thread-level parallelism support, to have the threads fill what they "think" are their private memories, but is actually our future ROM. In this special mode, yescrypt tells itself to expand the random read-writes phase to cover the running time that would have been spent on the unneeded random reads phase (which wouldn't have affected the memory region's contents). There are no RAMs yet during ROM initialization.

yescrypt ROM initialization

Second, the threads use first half of the ROM as their ROM and proceed with the random read-writes phase (skipping the sequential writes phase) on second half

```
+--------+--------+--------+--------+--------+--------+--------+
|  core  |11111111111111111111111111111111111111111111111111111|
+-------H+11111111111111111111111111111111111111111111111111111+
        #=11111111111111111111111111111111111111111111111111111|
         |11111111111111111111111111122222332232322222322222332+
        #=22323222232223232222222323222233223222323222223223332|
+-------H+22232322222222242442224224222224242224244224222224422+
|  core  |44422222422244222242242222444222222442424222222224222|
+--------+--------+--------+--------+--------+--------+--------+
```

We reuse yescrypt itself, along with its thread-level parallelism support, to have the threads fill what they "think" are their private memories, but is actually our future ROM. In this special mode, yescrypt tells itself to expand the random read-writes phase to cover the running time that would have been spent on the unneeded random reads phase (which wouldn't have affected the memory region's contents). There are no RAMs yet during ROM initialization.

yescrypt ROM initialization

Third, the threads use second half of the ROM as their ROM and proceed with both the sequential writes and the random read-writes phases on first half

```
+--------+--------+--------+--------+--------+--------+--------+
|  core  |55555555555555555555555555555555555555555555555555555|
+-------H+55555555555555555555555555555555555555556666666666666+
        #=66666666666666666666666666666666666666666666666666666|
         |66666666666666666666666666622222332232322222322222332+
        #=22323222232223232222222323222233223222323222223223332|
+-------H+22232322222222242442224224222224242224244224222224422+
|  core  |44422222422244222242242422224442222224424242222224222|
+--------+--------+--------+--------+--------+--------+--------+
```

We reuse yescrypt itself, along with its thread-level parallelism support, to have the threads fill what they "think" are their private memories, but is actually our future ROM. In this special mode, yescrypt tells itself to expand the random read-writes phase to cover the running time that would have been spent on the unneeded random reads phase (which wouldn't have affected the memory region's contents). There are no RAMs yet during ROM initialization.

yescrypt ROM initialization

Finally, the threads use first half of the ROM as their ROM and proceed with both the sequential writes and the random read-writes phases on second half

```
             +--------+--------+--------+--------+--------+--------+--------+
             |  core  |55555555555555555555555555555555555555555555555555555|
             +-------H+55555555555555555555555555555555555555556666666666666+
                     #=66666666666666666666666666666666666666666666666666666|
                      |66666666666666666666666666677777777777777777777777777+
                     #=77777777777777777777777777777777777777777777777777777|
             +-------H+77777777777778888888888888888888888888888888888888888+
             |  core  |88888888888888888888888888888888888888888888888888tag|
             +--------+--------+--------+--------+--------+--------+--------+
```

While we use sequential pass and thread numbers for illustrative purposes, the actual data will look highly random, and will be inefficient to partially recompute in a much smaller amount of memory. (Yes, ROM TMTO is a concern.)

The last 48 bytes of a ROM are its tag, which yescrypt checks for. This is to prevent inadvertent use of an incompletely or incorrectly initialized ROM.

yescrypt ROM upgrades

ROMs can be nested to allow for upgrades, where the old ROM remains around as part of the new ROM (rather than requiring more memory just for compatibility)

```
    +--------+--------+--------+--------+--------+--------+--------+--------+
    |                                   |                                   |
    +              old ROM              |                                   +
    |                               tag1|                                   |
    +-----------------------------------+              new ROM              +
    |                                                                       |
    +                                                                       +
    |                                                                   tag2|
    +--------+--------+--------+--------+--------+--------+--------+--------+
```

This is why the ROM tags are at the end: if they were at the beginning, they'd clash. The recommended upgrade step is 4x, but other upgrades are possible if existing hashes don't need to be upgraded (for which it has to be exactly 4x).

yescrypt high-level features

* Hash string encoding

  + Transparently encodes/decodes parameters, salt, and hash

* Hash (re-)encryption

  + Prevents offline password cracking if only the hash database is available

* Hash upgrades to higher cost settings without knowledge of passwords

  + 4x growth of RAM (and ROM if present) per upgrade, for 60%+ AT efficiency

  + If you can't afford 4x yet, then it's too early to upgrade (e.g., 2x upgrades would eventually result in only 33% AT efficiency)

* Client-side computation of almost final yescrypt hashes ("server relief") in a way allowing for a straightforward extension of SCRAM (RFC 5802)

  + Unfortunately, of limited use because of cache timing concerns

yescrypt cryptographic security

* Cryptographic security of the hash function (collision resistance, preimage and second preimage resistance) is based on that of SHA-256, HMAC, and PBKDF2

* The hash encryption feature uses provably secure Luby-Rackoff construction with SHA-256 as the PRF

* The known unfortunate peculiarities of HMAC and PBKDF2 are fully avoided in the way these primitives are used

* The rest of processing, while crucial for increasing the cost of password cracking attacks, may be considered non-cryptographic

  + There are entropy bypasses to final PBKDF2 step for both password and salt

  + For comparison, scrypt has such entropy bypass for password only

yescrypt timing safety

Timing safety was considered as part of yescrypt design, meaning that timing unsafe operations were avoided where practical. That said, unfortunately yescrypt is not cache timing safe. (FWIW, scrypt and bcrypt are also not cache timing safe. Argon2i and Catena are.)

This is a security vs. security trade-off: we could have cache timing safety, but per current knowledge it'd result in lower TMTO resistance, longer running time for same TMTO resistance, lower memory usage at same running time, and/or no or likely worse GPU attack resistance at low memory.

In practice, cache timing unsafety of password hashing functions is mitigated by use of unpredictable salts, which in most cases an attacker won't possess without also having the hashes. In yescrypt, both offline password cracking attacks and online cache timing attacks given a stolen/leaked password hash database can be further mitigated by use of the built-in encryption feature. Of course, this fails if the encryption key is also compromised. The situation is far from perfect, but that's the current trade-off and how it's resolved.

We recommend deployment on dedicated servers or at least dedicated NUMA nodes

yescrypt pros and cons

Pros

* Scalable from kilobytes to terabytes and beyond, and separately from a millisecond to eternity, while providing near-optimally high attack cost

* High attack cost across full range of attack hardware, from off-the-shelf to specialized

* Feature-rich. Can also compute classic scrypt (shared codebase).

Cons

* Complex

  + OK for large-scale deployments, where it's a small percentage of total complexity e.g. of a user authentication microservice
  + A con and risk for smaller deployments and third-party implementations
    + A future yescrypt-lite and/or popular library support might help

* Cache timing unsafe

yescrypt demo setup

On 2x E5-2670 (16 cores, 32 threads) with 128 GiB RAM (8x DDR3-1600)

* 112 GiB ROM initialization takes half a minute (on server bootup)
  + Initialization algorithm discourages recomputation on smaller machines
* ROM is maintained in a SysV shared memory segment
  + This is one good way to do it, although yescrypt API does not mandate it
* Thus, there's no delay in our demo "authentication service" (re)start
  + Important for ease of system administration, and to minimize downtime
* yescrypt is passed a pointer to the ROM via an appropriate API

yescrypt demo

Linux configuration: a bit more than 112 GiB (120 GB) in "huge pages"

```
[root@super ~]# sysctl -w vm.hugetlb_shm_group=500
vm.hugetlb_shm_group = 500
[root@super ~]# sysctl -w kernel.shmmax=120259084288
kernel.shmmax = 120259084288
[root@super ~]# sysctl -w kernel.shmall=29622272
kernel.shmall = 29622272
[root@super ~]# sysctl -w vm.nr_hugepages=57856
vm.nr_hugepages = 57856
[root@super ~]# free
             total       used       free     shared    buffers     cached
Mem:     132125032  121733836   10391196        816     263560    1524240
-/+ buffers/cache:  119946036   12178996
Swap:            0          0          0
```

yescrypt demo

112 GiB (120 GB) ROM initialization (31 seconds)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 time ./initrom 112 1
r=14 N=2^10 NROM=2^26
Will use 117440512.00 KiB ROM
         1792.00 KiB RAM
Initializing ROM ... DONE (8f39cc22)
'$y$j7B5N$LdJMENpBABJJ3hIHjB1Bi.$4QWkjJaK.MA7IzRxHzafBUmnMMhFYUFvXQ11tVxmNG.'
380.40user 47.68system 0:30.95elapsed 1382%CPU (0avgtext+0avgdata 11616maxresiden
t)k
0inputs+0outputs (0major+58683minor)pagefaults 0swaps
```

yescrypt demo

112 GiB ROM, 1.75 MiB RAM password hashing test (over 10000 hashes/second)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 ./userom 112 1
r=14 N=2^10 NROM=2^26
Will use 117440512.00 KiB ROM
         1792.00 KiB RAM
Plaintext:
'$y$j7B5N$LdJMENpBABJJ3hIHjB1Bi.$4QWkjJaK.MA7IzRxHzafBUmnMMhFYUFvXQ11tVxmNG.'
Encrypted:
'$y$j7B5N$LdJMENpBABJJ3hIHjB1Bi.$PADH1GUHcX9IN7aHafrRR7b8qfBrey1CM2/vnEUyXT1'
Benchmarking 1 thread ...
647 c/s real, 651 c/s virtual (1023 hashes in 1.58 seconds)
Benchmarking 32 threads ...
10438 c/s real, 330 c/s virtual (15345 hashes in 1.47 seconds)
```

We set CPU affinity on these tests. Results are similar, but somewhat less stable without that setting. A service using yescrypt could do this too, but even if it doesn't it'd be likely impacted less than the "userom" program is, since OpenMP needs thread synchronization while user authentication wouldn't.

yescrypt demo

```
Password hashing microservice using yescrypt at 112 GiB ROM, 1.75 MiB RAM -
status page (when testing, non-production, hence so many "new hash requests")

Service uptime    6068 seconds (12.34 since last status)

Hashing requests  62168081, 10245/s (126455, 10248/s)
 OK               62168081, 100.00%, 10245/s (126455, 100.00%, 10248/s)
 Error            0, 0.00%, 0/s (0, 0.00%, 0/s)
  Bad JSON        0, 0.00% of all errors (0, 0.00%)
  Failed/refused  0, 0.00% of all errors (0, 0.00%)

New hash requests 62092545, 99.88%, 10233/s (126300, 99.88%, 10235/s)
 New hash OK      62092545, 100.00% of new hash requests
 New hash error   0, 0.00% of new hash requests

Verify requests   75536, 0.12%, 12/s (155, 0.12%, 13/s)
 Verify OK        75536, 100.00% of verify requests
 Verify error     0, 0.00% of verify requests
```

yescrypt demo

112 GiB ROM, 3.5 MiB RAM password hashing test (over 5000 hashes/second)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 ./userom 112 3
r=14 N=2^11 NROM=2^26
Will use 117440512.00 KiB ROM
         3584.00 KiB RAM
Plaintext:
'$y$j8B5N$LdJMENpBABJJ3hIHjB1Bi.$v2V7ppNIzgufHMSnNLa23Ir0ZbPp5xPS1ob5jOKiJi1'
Encrypted:
'$y$j8B5N$LdJMENpBABJJ3hIHjB1Bi.$qsqs13r6dL1.JS/r9tXMjBponmwcTS43bXn6.DoBVH1'
Benchmarking 1 thread ...
333 c/s real, 333 c/s virtual (511 hashes in 1.53 seconds)
Benchmarking 32 threads ...
5110 c/s real, 160 c/s virtual (7665 hashes in 1.50 seconds)
```

yescrypt demo

112 GiB ROM, 14 MiB RAM password hashing test (over 1000 hashes/second)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 ./userom 112 14
r=14 N=2^13 NROM=2^26
Will use 117440512.00 KiB ROM
         14336.00 KiB RAM
Plaintext:
'$y$jAB5N$LdJMENpBABJJ3hIHjB1Bi.$Qrspo8omJ7gmU.M5OQ7xLwnyeJ023mLxWwJ7tWCb1XB'
Encrypted:
'$y$jAB5N$LdJMENpBABJJ3hIHjB1Bi.$a2Q9Zf.1E62cIXE.ziUvSIj6SGBgJ3JLv/cDQ99InW4'
Benchmarking 1 thread ...
81 c/s real, 81 c/s virtual (127 hashes in 1.55 seconds)
Benchmarking 32 threads ...
1076 c/s real, 35 c/s virtual (1905 hashes in 1.77 seconds)
```

yescrypt demo

Here's what the SysV shm segment looked like, and how to destroy it

```
[solar@super yescrypt-0.9.19]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x7965730a 65536      solar      640        120259084288 0



[solar@super yescrypt-0.9.19]$ ipcrm -M 0x7965730a
[solar@super yescrypt-0.9.19]$
```

yescrypt demo

No ROM, 2 MiB RAM password hashing test (over 10000 hashes/second)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 ./userom 0 2
r=8 N=2^11 NROM=2^0
Will use 0.00 KiB ROM
         2048.00 KiB RAM
Plaintext:
'$y$j85$LdJMENpBABJJ3hIHjB1Bi.$bAfr8bOshRhNY74kqJ7IEJD9fzS2/JRu6jYSI5oSKpD'
Encrypted:
'$y$j85$LdJMENpBABJJ3hIHjB1Bi.$200Zp58Dexbc3pm04wly3CMe9N9Q2eXORYFKpdejew4'
Benchmarking 1 thread ...
691 c/s real, 691 c/s virtual (1023 hashes in 1.48 seconds)
Benchmarking 32 threads ...
10960 c/s real, 344 c/s virtual (15345 hashes in 1.40 seconds)
```

yescrypt demo

No ROM, 16 MiB RAM password hashing test (over 1000 hashes/second)

```
[solar@super yescrypt-0.9.19]$ GOMP_CPU_AFFINITY=0-31 ./userom 0 16
r=8 N=2^14 NROM=2^0
Will use 0.00 KiB ROM
         16384.00 KiB RAM
Plaintext:
'$y$jB5$LdJMENpBABJJ3hIHjB1Bi.$pQdX./.PtwBB6SLje6nsfOf2oT5RaDG/tZbatmUEXA1'
Encrypted:
'$y$jB5$LdJMENpBABJJ3hIHjB1Bi.$tR0cZxQ7pdKu1t2K1Pd.Ne/LH5dJEKICeosHSnxYOA6'
Benchmarking 1 thread ...
88 c/s real, 88 c/s virtual (127 hashes in 1.44 seconds)
Benchmarking 32 threads ...
1024 c/s real, 32 c/s virtual (1905 hashes in 1.86 seconds)
```

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
Password Hashing Competition

Contact information

e-mail
Solar Designer <solar@openwall.com>

Twitter
@solardiz @Openwall

website
http://www.openwall.com