Ω

Omega Crypt (ocrypt)

Brandon Enright
bmenrigh@brandonenright.net
http://www.brandonenright.net/ocrypt/
Version 1.0; 2014-03-31

Abstract: Omega Crypt (ocrypt) is a novel password hashing and key derivation function designed in the general spirit of sequential memory-hard functions. Tunable memory usage and time (computation) parameters give ocrypt many of the anti-ASIC and anti-FPGA properties of scrypt. ocrypt is designed to defeat stream processing architectures like GPUs by using the output of a stream cipher to drive secure data-dependent branching while carefully avoiding the obvious side-channel issues with data-dependent execution.

Background and Motivation

Password hashing has long been the most common way to store and protect a user's password against attack while still allowing verification of the user. By storing a one-way transformation of a password the only way to determine what the corresponding password is for the hash is to "crack" it -- guess likely password candidates, apply the one-way transformation, and check to see if the result matches the hash. This property that attackers must guess candidate password and check them is highly desirable which is why one-way cryptographic hashing is used in the first place. It has become clear though that using one-way functions isn't adequate[1] for two main reasons:

1. Users choose poor, easy-to-guess passwords
2. There is a significant asymmetry between the resources of attackers and defenders

The former issue isn't technical and can't be fixed with technology so password hashing schemes have tended to focus on the latter.

The main three password hashing contenders are PBKDF2, bcrypt, and scrypt[2] and each has its pros and its cons. Beyond traditional tricks like salts and lots of iterations to use CPU time, neither PBKDF2 or bcrypt significantly address the computational asymmetry between defenders and attackers. scrypt attempts to negate some of the hardware advantages attackers usually have by making use of large amounts of memory in a random.

The current state of password hashing security is that defenders are mostly confined to using general purpose CPUs and memory on commodity computing hardware. Attackers have no such limitations and will use whatever tool is best for the job -- usually a GPU -- to try guesses faster or in parallel or both. GPUs (as well as many ASIC and FPGA designs) use a computing model known as single instruction, multiple data (SIMD) or (usually) very wide SIMD architectures called stream processing. Stream processing works best when there are many, often thousands of computing units that can be all kept executing lock-step with one another the same instruction across processors at the same time all operating on a different set of data. The stream processing computing model often works well for attackers even when the hash algorithm doesn't have any internal parallelism because the attacker can instead achieve parallelism by computing many discrete hashes at the same time in lock-step.

The approach scrypt uses to attack stream processing is to make memory accesses the bottleneck. Unfortunately due to a built-in time-memory tradeoff as well as the sheer amount and speed of memory available on modern GPUs, scrypt isn't as effective at defeating stream processing as a modern scheme should be.

To reduce the asymmetry between attackers and defenders an algorithm needs to go after attacker's weaknesses and negate their strengths. bcrypt does a good job of using arbitrary amounts of computing power but this hurts the defender equally. scrypt attempts to use

arbitrary amounts of computing power and cause bottlenecks in memory but modern GPUs are starting to negate the effectiveness of memory usage as the only defense strategy. The stream processing computing model itself must be negated. Omega Crypt is a proposal to do just that.

Algorithm Summary

The core idea ocrypt is designed around is the realization that data-dependence defeats the fundamental computing model of stream processing and very wide SIMD architectures. If discrete processing units can't be kept executing in lock-step with one another, most GPUs as well as many ASIC and FPGA designs are severely hampered, if not rendered completely useless.

Data-dependent branching is not trivial though. Side channel and timing attacks due to data-dependence are well-known and have broken many real-world systems. To avoid the drawbacks of data-dependant branching ocrypt only relies on the output of a stream cipher and never any user input or ocrypt parameters such as the password, salt, personalization key, or the lengths of these parameters.

At a high level, ocrypt takes user input and algorithm parameters and pad them to a fixed size and hashes them (with cubehash)[3] to derive a key for a stream cipher (ChaCha)[4]. The stream cipher output is used to initialize an arbitrarily large (configurable) block of memory. Once initialized, the output of the stream cipher is used in a loop to select one of several branches, each of which manipulate the memory block differently. The specific memory addresses accessed are also guided by stream cipher output and a carry value is used, changed, and carried over from each iteration to enforce a sequential data dependence between the iterations. Once this is executed an arbitrary many times (configurable), the whole data block is fed into cubehash for output as the result.

Algorithm Specification

Algorithm ocrypt(P, S, K, T, M, L)

Input:

P	Password of length P_l bytes
S	Salt of length S_l bytes
K	Personalization key of length K_l bytes
L	The length of the the hash output in bytes
T, M	Parameters to control Time usage and Memory usage

Output:

(H_0 … H_L-1) bytes

Restrictions:

P, S, K The length of these inputs must be <= 255 bytes
L The output length must be one of {16, 20, 28, 32, 48, 64}
T, M The cost parameters must be in the range [0 … 14] inclusive

Steps:

1. Pad P, S, K to a length of 255 with 0x00 bytes
2. Concatenate {P, P_l, S, S_l, K, K_l, L, T, M} to create a parameter string Q of exactly 771 bytes
3. Hash Q with cubehash160+16/32+160-256 to derive a 256-bit key C_k
4. Initialize ChaCha8 with the key C_k and a 64-bit IV 0x0000000000000000. The output of ChaCha8 is a stream of bytes and little-endian integers are filled, they are filled MSB first so that the byte-order of an encrypted integer in memory is the same byte-order of ChaCha8 output.
5. Allocate and zero-fill 2^(17 + M) 64-bit words as a state array A. 17 is the base memory cost parameter and corresponds to 1 MiB of memory. When accessed as an array A[] the 64-bit words are treated as little-endian integers. When accessed as a stream of bytes A there is no endianness. The length 2^(17 + M) is A_l and the value A_l - 1 is the bitmask A_m
6. Set the first 771 bytes of A to the parameter string Q
7. Encrypt A[] 64-bits at-a-time from the output of ChaCha8
8. Initialize a 64-bit carry integer R with ChaCha8
9. For 2^(17 + T) iterations (where 17 is the base time cost) loop:

   9_a: Set B to 1-byte of ChaCha8 output and zero all but the lower to bit of B to produce a B in the range [0 … 3]

   9_b: if B == 0 do:
      Set TAD_a to 4-bytes of ChaCha8 & A_m
      Set TVAL_a to 8-bytes of ChaCha8
      A[TAD_a] += R
      R ^= TVAL_a

   9_c: if B == 1 do:
      Set TAD_a to (4-bytes of ChaCha8 XOR 0x0a1b2c3d) & A_m
      Set TVAL_a to 8-bytes of ChaCha8
      A[TAD_a] ^= R
      R += TVAL_a

   9_d: if B == 2 do:
      Set TAD_a to (4-bytes of ChaCha8 XOR 0xfedc0123) & A_m
      Set TAD_b to (4-bytes of ChaCha8 XOR 0xfedc0123) & A_m
      Set TVAL_a to 8-bytes of ChaCha8

Set TVAL_b to 8-bytes of ChaCha8
A[TAD_a] ^= TVAL_a
A[TAD_b] += (TVAL_b ^ R)
R ^= A[R & A_m]

9_e: if B == 3 do:
Set TAD_a to (4-bytes of ChaCha8 XOR 0x76543210) & A_m
Set TVAL_a to 8-bytes of ChaCha8
Set TVAL_b to 8-bytes of ChaCha8
A[A[TAD_a] & A_m] += (R ^ TVAL_a)
R += (A[TAD_a] ^ TVAL_b)

10. Hash the full state array bytes A with cubehash16+8/64+320-(L * 8) where (L * 8) is the length of the hash output in bits to produce the final hash output H_0 .. H_L-1

Design Rationale

Although ocrypt is defined with cubehash and ChaCha8 as the cryptographic primitives, it is the overall construction that provides the computational and memory difficulty, not the specific hash and stream cipher used. cubehash was chosen for it's extreme simplicity and flexibility. ChaCha8 was chosen for its extreme simplicity and speed.

In step 1, 2, and 3 the input parameter are padded and hashed in such a way that ocrypt does not leak the length of any of the parameters in the hashing step and the length of the parameters is appended to the padded version of each parameter to eliminate trivial collisions like "pass\0" and "pas\0\0". This also makes the salt 0x00 different than 0x0000, etc. Also, all parameters are used to derive the ChaCha8 key so that changing any input, salt, password, or otherwise completely changes the ChaCha8 ciphertext output stream to ensure the branching sequence of every ocrypt instance is unique (within the limits of a 256-bit key).

Step 6 is intended to carry over every bit of input into the mixing phase because ChaCha8 is initialized with only 256-bits worth of initial parameters. Step 7 protects the initial parameters and sets up the remaining state array for mixing.

Step 8 sets up the carry value for step 9. Step 9 is intended to branch unpredictably based on the output of ChaCha8. Each sub-step 9 option depends on the carry value to enforce linearity. Each sub-step 9 option also reads and writes at least one pseudo-random state-array location. Each sub-step 9 option uses 32-bits from ChaCha8 as an array index so to prevent prefetching, the value is XORed differently for three of the options. Of the four sub-step 9 options, three of them extract different amounts of ciphertext output from ChaCha8 to prevent pre-caching of branch or address values for future iterations.

Step 10 compresses the full state array into an L-byte output so that the result of every previous step is factored into the final output.

Efficiency Analysis

ocrypt takes two cost parameters, a time cost T and a memory cost M. The intention is that these parameters will give the user of ocrypt full control over the overall cost of the algorithm. The choice of cubehash and ChaCha8 were made to allow implementations to be as efficient as possible on modern CPUs. The only inefficient-by-design portion of ocrypt is just the ChaCha8 manipulation of the state array.

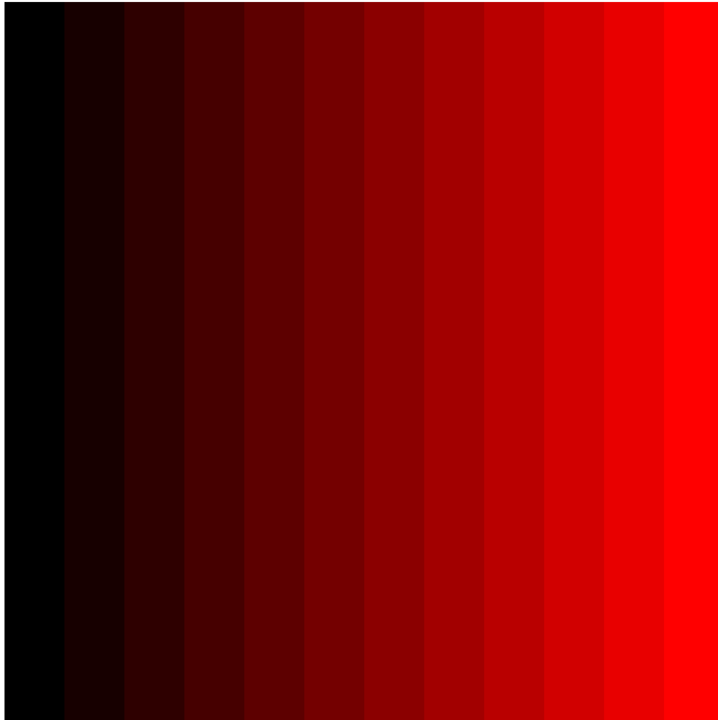
The ocrypt manipulation of the state array is not meant to be efficient on any platform. Very little work is done between random branches which prevents deep pipelining, SIMD, or stream processing from being useful. The data-dependant branching is meant to “level the playing field” across all main forms of computing down to a single CPU worth of sequential work. The high memory usage and use of 64-bit arithmetic is meant to increase the cost * time of using GPUs, ASICS, and FPGAs to equal or above the cost * time of general-purpose computing.

The reference implementation of ocrypt is not particularly efficient and significant improvements could be made to the cubehash and ChaCha8 implementations so that they become a negligible portion of the running time.

When T and M costs are set to 0 ocrypt uses 1 MiB of memory and performs 2^17 = 131072 pseudo-random branches. In the current (inefficient) implementation that takes about 30ms of CPU time. Better cubehash and ChaCha8 implementations should be able to drop this to 10ms or below.

The T and M parameters are both log2 values of the intended time and memory usage so increasing T by 1 should roughly double the CPU time needed and increasing M by 1 exactly doubles the memory usage. Initializing the memory with ChaCha8 and then later hashing it with cubehash isn’t free though, so in the current implementation increasing M by 1 also approximately doubles the total time needed. For larger values of T, increasing M has less overall effect and if cubehash and ChaCha8 are improved, M will have even less effect on the total runtime.

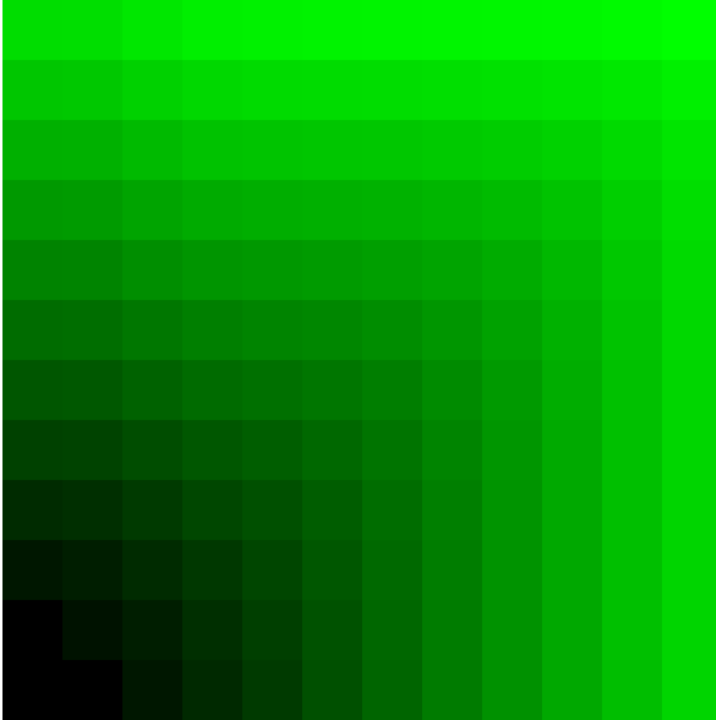
Effect of M and T on memory (log value):



T ranges from 0 to 11 from bottom to top on the Y axis
M ranges from 0 to 11 from left to right on the X axis

T has no effect on memory usage and memory usage changes exponentially (doubles) with M.

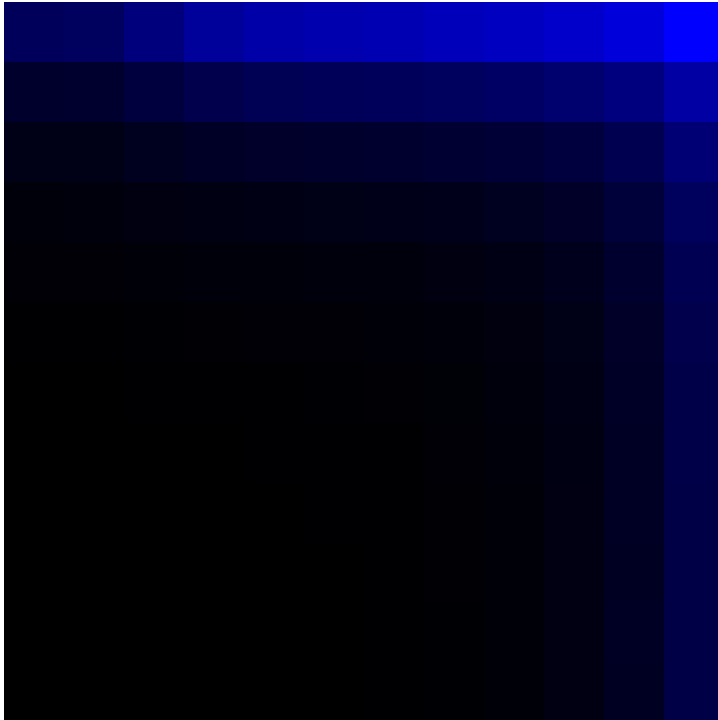
Effect of M and T on time usage (log value):



T ranges from 0 to 11 from bottom to top on the Y axis
M ranges from 0 to 11 from left to right on the X axis

T and M have roughly equal effects on total runtime (shown as the log of the time) making the time nearly symmetrical along the diagonal. Increasing T has a slightly greater impact on total time but the effect can't easily be seen.

Effect of M and T on time usage (linear value):



T ranges from 0 to 11 from bottom to top on the Y axis
M ranges from 0 to 11 from left to right on the X axis

T and M have roughly equal effects on total runtime (shown as the the actual value, scaled) making the time nearly symmetrical along the diagonal. Increasing T has a slightly greater impact on total time but the effect can't easily be seen. This is the same data as the previous image in green but without the log of the total time.

Security Analysis

ocrypt is intended to be cryptographically secure. n-bit ocrypt output should have the same collision resistance, preimage resistance, and other cryptographic properties of n-bit hash functions.

The padding scheme for the password, salt, and key, as well as other parameters is done in a way to prevent length-extension and the usage of cubehash160+16/32+160-256 for the ChaCha8 key derivation benefits from the strength, analysis, and community confidence in cubehash. ChaCha8 is a well-known, secure stream cipher whose output, after mixing, is fed into another instance of cubehash. Weaker but significantly faster parameters were chosen for the final cubehash usage (cubehash16+8/64+320-N). The design of ocrypt prevents an attacker from having any direct or meaningful control over the input to the final cubehash should an attack on cubehash16+8/64+320-N be found.

The data-dependant branching and manipulation of the state array is not intended to provide any cryptographic security. The particular manipulations for each of the four options were chosen not for cryptographic reasons but instead for variety and to make the state array manipulation approximate a sequential memory hard function.

Due to the usage of data-dependant branching in ocrypt, a significant amount of information about the output of ChaCha8 is leaked in both memory access patterns and timing. All published analysis suggests that ChaCha8 is not susceptible to any known-ciphertext attacks and that leaking information about the output of ChaCha8 does not leak any information about the key used. In the event that a major weakness is found in ChaCha8 (or if it were to be replaced by a much less secure stream cipher), the key being leaked is the output of cubehash.

It follows from the significant ChaCha8 output side channel of ocrypt that it is trivial to produce a “fingerprint” of activity that could uniquely differentiate each parameter set input to ocrypt. This could, for example, uniquely identify when a particular user’s password is being verified. Without other information not derived from ocrypt it would not be possible to associate a given parameters fingerprint to a specific user. No password, salt, or personalization key information, including length, are leaked by the data-dependant branching.

Suggested Future Tweaks

Neither cubehash or ChaCha8 make use of 64-bit machine words. Both can be naturally extended to 64-bit words (word rotation values would need to be adjusted) without significantly affecting the security of either. Using 64-bit words would better use modern CPUs and disadvantage current GPU, ASIC, and FPGA technology.

ChaCha8 is overly conservative for the role it provides in ocrypt. It could be reduced to 4 rounds which would reduce the effect of the M on the total runtime.

cubehash16+8/64+320-N is still probably rather conservative and could be reduced. The attacker has almost no control over the input to the final cubehash so something like cubehash16+4/96+320-N should be fine.

The password limitation of 255 characters was chosen because “255 should be enough for anyone” but it could be naturally extended to much larger ranges like 65535 easily.

Code, Supplementals, Limitations

A Known-Answer-Test (KAT) for ocrypt across many different parameters has been provided in KAT_ocrypt.txt and can be generated by building ocrypt_genKAT.c
To use the PHS specification of ocrypt (which doesn’t accept a personalization key) use phs.h but to use the full ocrypt with key use ocrypt.h

The reference implementation of ocrypt is sensitive to endianness and the KAT file was generated on a little-endian machine. The cubehash implimentation will need to be changed to something endian-aware and a few locations in ocrypt will need to be adjusted for big-endian machines too.

The ChaCha8 implementation was taken from the nettle project (https://github.com/secworks/nettle) and chacha-wrapper.h is used to make the nettle implementation more friendly to the specific way ocrypt uses ChaCha8.

Intellectual property statement

Omega Crypt has been placed in the public domain and is and will remain available worldwide on a royalty free basis. I am unaware of any patent or patent application that covers the use or implementation of Omega Crypt.

Statement of hidden weaknesses

There are no deliberately hidden weaknesses (backdoor, etc.) or ill intent in or with the design of Omega Crypt.

Acknowledgements

Thanks to Alexander Peslyak (Solar Designer) for leading the way.
Thanks to Jens Steube (atom) for showing us what’s possible.
Thanks to Colin Percival, your contributions fill a huge void.
Thanks to Daniel J. Bernstein, the world needs more cryptographers of your caliber.

References

[1] Password security: past, present, future
http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/

[2] Stronger Key Derivation via Sequential Memory-Hard Functions
https://www.tarsnap.com/scrypt/scrypt.pdf

[3] Introduction to Cubehash
http://cubehash.cr.yp.to/

[4] The ChaCha family of stream ciphers
http://cr.yp.to/chacha.html