

# CubeHash specification (2.B.1)

Daniel J. Bernstein \*

Department of Computer Science  
University of Illinois at Chicago  
Chicago, IL 60607-7045  
cubehash@box.cr.yp.to

**Algorithm summary.** CubeHash $r/b-h$  produces an  $h$ -bit digest of a variable-length message. The digest depends on two tunable parameters  $r, b$  that allow the selection of a range of security/performance tradeoffs. CubeHash is defined for each  $h \in \{8, 16, 24, \dots, 512\}$ , each  $r \in \{1, 2, 3, \dots\}$ , each  $b \in \{1, 2, 3, \dots, 128\}$ , and each message length from 0 bits through  $2^{128} - 1$  bits. Third-party cryptanalysts are encouraged to start with large values of  $b$  and small values of  $r$ .

**Algorithm specification.** CubeHash $r/b-h$  has five major steps:

- Initialize a 128-byte (1024-bit) state as a function of  $(h, b, r)$ .
- Convert the input message into a padded message. The padded message consists of one or more  $b$ -byte blocks.
- For each  $b$ -byte block of the padded message: xor the block into the first  $b$  bytes of the state, and then transform the state invertibly through  $r$  identical rounds.
- Finalize the state.
- Output the first  $h/8$  bytes of the state.

Initialization works as follows. The 128-byte state is viewed as a sequence of 32 4-byte words  $x_{00000}, x_{00001}, x_{00010}, x_{00011}, x_{00100}, \dots, x_{11111}$ , each of which is interpreted in little-endian form as a 32-bit integer. The first three state words  $x_{00000}, x_{00001}, x_{00010}$  are set to the integers  $h/8, b, r$  respectively. The remaining state words are set to 0. The state is then transformed invertibly through  $10r$  identical rounds.

Padding works as follows. Append a 1 bit to the input message; then append the minimum possible number of 0 bits to reach a multiple of  $8b$  bits. The bits in a byte are first 128, then 64, then 32, then 16, then 8, then 4, then 2, then 1.

(History suggests that some, perhaps most, implementations will be restricted to byte-aligned inputs. These implementations can simply append a 128 byte and then the minimum possible number of 0 bytes to reach a multiple of  $b$  bytes.)

Note that this padding does not require separate storage of the message length, the block to be processed, etc. The implementor can simply store a single integer between 0 and  $8b$  to record the number of bits processed so far within the current block.

---

\* The author was supported by the National Science Foundation under grant ITR-0716498. Date of this document: 2008.10.28.

Finalization works as follows. The integer 1 is xored into the last state word  $x_{111111}$ . The state is then transformed invertibly through  $10r$  identical rounds.

Each round has the following ten steps:

- Add  $x_{0jklm}$  into  $x_{1jklm}$  modulo  $2^{32}$ , for each  $(j, k, l, m)$ .
- Rotate  $x_{0jklm}$  upwards by 7 bits, for each  $(j, k, l, m)$ .
- Swap  $x_{00klm}$  with  $x_{01klm}$ , for each  $(k, l, m)$ .
- Xor  $x_{1jklm}$  into  $x_{0jklm}$ , for each  $(j, k, l, m)$ .
- Swap  $x_{1jk0m}$  with  $x_{1jk1m}$ , for each  $(j, k, m)$ .
- Add  $x_{0jklm}$  into  $x_{1jklm}$  modulo  $2^{32}$ , for each  $(j, k, l, m)$ .
- Rotate  $x_{0jklm}$  upwards by 11 bits, for each  $(j, k, l, m)$ .
- Swap  $x_{0j0lm}$  with  $x_{0j1lm}$ , for each  $(j, l, m)$ .
- Xor  $x_{1jklm}$  into  $x_{0jklm}$ , for each  $(j, k, l, m)$ .
- Swap  $x_{1jkl0}$  with  $x_{1jkl1}$ , for each  $(j, k, l)$ .

That's it.

**Application specification.** CubeHash- $h$  can be used in the same contexts as SHA- $h$ . Applications such as HMAC that pad to a full block of SHA- $h$  input are required to pad to a full minimal integral number of  $b$ -byte blocks for CubeHash $r/b$ - $h$ .

**Parameter recommendations.** I recommend

- CubeHash8/1-224,
- CubeHash8/1-256,
- CubeHash8/1-384, and
- CubeHash8/1-512;

i.e.,  $r = 8$  and  $b = 1$ , independent of the message-digest size.

I see no way to attack  $r = 4$ , even with rather large  $b$ , so I think that the half-speed choice  $r = 8$  leaves a comfortable security margin. I also see no way to attack  $b = 2$ , even with rather small  $r$ , so I think that the half-speed choice  $b = 1$  leaves a comfortable security margin. In fact, I think that CubeHash8/1 is much, much, much more conservative than necessary.

Justification for recommending CubeHash8/1: For most applications of hash functions, speed simply doesn't matter. High-volume network protection with HMAC is sometimes cited as an exception, but anyone who really cares about speed shouldn't be using HMAC anyway; other MACs are faster and inspire more confidence.

What about the occasional applications where hashing speed *does* matter? If, after third-party cryptanalysis, the community is convinced that much faster CubeHash $r/b$  choices are perfectly safe, then I expect those choices to be considered in speed-oriented applications. For the same reason, I expect NIST to scientifically compare the speeds of different hash-function families by comparing the speeds of the fastest unbroken members of those families—rather than biasing the comparison according to the speculative initial recommendations of the hash-function designers.

According to NIST, this document should “provide any bounds that the designer feels are appropriate for the parameter, including a bound below which the submitter expects cryptanalysis to become practical.” I hereby declare that, for every  $r/b$  below  $1/112$ , I expect cryptanalysis of CubeHash $r/b$  to become practical (if it is not already practical). Note that there is no requirement to comment on—and I am not commenting on—the practicality of cryptanalysis *above* the specified bound.

**Design rationale and explanation.** CubeHash does not include block counters, block randomizers, etc. The entire argument for those complications is that state collisions are generically easy to find (compared to the desired preimage security); but this argument breaks down when the state size is large enough. A generic attack using an unimaginable  $2^{400}$  operations has an utterly negligible chance of finding a collision in CubeHash’s 1024-bit state. In the absence of state collisions, the state determines the entire previous message, so it determines the block counter, any block randomizer included earlier in the message, etc.

One might argue that there is no *loss* in security from including block counters etc. However, if there is no *gain* in security, then it makes more sense to devote the same resources to confidence-building measures such as increasing  $r$  or decreasing  $b$ .

Similarly, there is no need to break any of the symmetries of the CubeHash round. The initial CubeHash states do not have any of those symmetries, and if  $b$  is not too large then the attacker does not have enough control to force a symmetric state or a pair of states related by those symmetries. I wouldn’t be surprised if an asymmetric round allows slightly smaller  $r/b$ , but it seems to me that this benefit would be outweighed by the cost in efficiency of each round.

The basic operations in CubeHash—32-bit add, 32-bit xor, and fixed-distance 32-bit rotation—have the virtue of taking constant time on typical CPUs; most implementations will avoid all software-level side-channel leaks. For comparison, most AES software implementations leak their keys through cache timing, prompting Intel to add constant-time AES instructions to its future “Westmere” CPUs.

The CubeHash round is designed to spread changes very quickly through the 1024 state bits. The additions and xors directly spread  $x_{ijklm}$  changes along the  $i$  axis of the  $(i, j, k, l, m)$  hypercube; the first swap prevents changes from being confined along the  $j$  axis; the second, third, and fourth swaps do the same for the  $l, k,$  and  $m$  axes respectively; the rotations move changes through the 32 bit positions within each word, in particular diffusing changes from high bits to low bits. Alternation between add and xor is a common technique to maximize the apparent randomness of carry bits.

The rotation distances 7, 11 in CubeHash were chosen as follows. Notice that, modulo 32,  $\{0, 7\} + \{0, 11\} + \{0, 7\} + \{0, 11\} = \{0, 4, 7, 11, 14, 18, 22, 25, 29\}$ , with a maximum gap of 4, which is obviously best possible;  $\{0, 7\} + \{0, 11\} + \{0, 7\} = \{0, 7, 11, 14, 18, 25\}$ , with a maximum gap of 7, which is best possible given the 4;  $\{0, 7\} + \{0, 11\} = \{0, 7, 11, 18\}$ , with a maximum gap of  $0 - 18 = 14$ , which is best possible given the  $(4, 7)$ . The only other ways to achieve  $(4, 7, 14)$  are with

rotation distances 7, 21 or 11, 25 or 21, 25; I took the first choice 7, 11. Carries also provide some diffusion across nearby bit positions, so the exact choice of rotation distances doesn't seem terribly important.

**Security arguments and preliminary analysis.** “Provable security” can be a helpful guide to cryptanalysis, allowing cryptanalysts to focus their energies on the potentially breakable components of a hash function. For example, most hash functions are built as modes of operation of compression functions; a sufficiently powerful security proof for a mode of operation lets cryptanalysts focus on the compression function, confident that the mode per se has no problems. Building confidence in the hash function then boils down to building confidence in the compression function, hopefully a simpler task.

CubeHash takes a different approach to building confidence: the entire hash function is very simple and easy to understand, in fact simpler than most compression functions. CubeHash is not built from a lower-level primitive, so there are no reduction proofs, but I don't see this as a disadvantage; the real question is not how much work the cryptanalyst can skip, but how much work the cryptanalyst still has to do.

CubeHash offers an array of fast variants as targets for cryptanalysis. The main security argument for any particular CubeHash $r/b$  will always be that the parameters  $(r, b)$  are beyond the limits of state-of-the-art attacks.

Preliminary analysis of collision-finding, preimage-finding, second-preimage-finding, length-extension attacks, multicollision attacks, etc.: These attacks appear to be extremely difficult. See also the separate “attack analysis” document.