StreamHash Algorithm Specifications and Supporting Documentation

Michal Trojnara

October 14, 2008

Abstract

The paper contains the algorithm specifications and supporting documentation for the StreamHash SHA-3 candidate.

1 Algorithm Specifications

1.1 StreamHash State

StreamHash state structure consists of:

- A vector of 32-bit values to hold the state for all processed bytes, hereafter referred to as the state vector;
- The value of remaining bits in the last input data byte, if it is not full; and
- The number $\{0, 1, \ldots 7\}$ of remaining bits in the last input data byte.

The length of the state vector is equal to the message digest size divided by 32, i.e. 7 for 224-bit digest, 8 for 256-bit digest, 12 for 384-bit digest, and 16 for 512-bit digest.

1.2 Initialization

At initialization the aforementioned state variables are set to zero.

1.3 State Update

For each input data byte the state update function updates each state vector value XORing it with an S-BOX output. The S-BOX index is computed as

$$LSB(state_i) \oplus b \oplus i$$

, where i is the state vector index, b is the input data byte value, and $\oplus$ is the bitwise XOR operation.

The resulting formula to update a state vector value for index i is:

$$state_i \coloneqq state_i \oplus S\text{-}BOX[LSB(state_i) \oplus b \oplus i]$$

Any remaining input data bits (for input size not being a multiple of 8 bits), and the number of these bits are both saved within the state structure.

1.4 Finalization

1.4.1 Updating State Vector with Remaining Bits

Two additional bytes are processed with State Update function, as if they were appended to the previously processed data:

- Any remaining bits as defined above; and
- The number of remaining bits.

Any unused bits in the remaining bits are set to zero.

1.4.2 Updating State Vector with Chosen State Bits

A vector is than built from lower 16 bits of each state vector word (in high-endian byte order). State is updated with this vector to provide resistance against length-extension attacks.

1.4.3 Diffusing the State Vector

For each state vector index i the state vector is updated as follows:

$$state_{(i+1) \bmod n} \coloneqq state_{(i+1) \bmod n} + state_i$$

, where n is the state vector size, and $+$ is addition modulo 2^{32}.

The above loop is performed three times.

1.4.4 Copying the State Vector into the Returned Vector

The state vector of 32-bit integers is copied into the returned vector of 8-bit bytes utilizing high-endian byte order.

1.4.5 Diffusing the Returned Vector

A simple, reversible transformation of the state vector into the returned vector is performed. For each returned vector index i the returned vector is updated as follows:

$$output_{(i+1) \bmod n} \coloneqq output_{(i+1) \bmod n} + output_i$$

, where n is the returned vector size, and $+$ is addition modulo 2^{32}.

1.5 Numerical Examples

Tables 1-6 illustrate the computations described above for the StreamHash-256 variant.

1.6 Building S-BOX

StreamHash S-BOX is based on AES S-BOX. The formula is to compute value 32-bit S-BOX value for index i is:

$$s(i) \vee (s(s(i)) \ll 8) \vee (s(s(s(i))) \ll 16) \vee (s(s(s(s(i)))) \ll 24)$$

, where $s(i)$ is the value of 8-bit AES S-BOX for index i, $\vee$ is the bitwise OR operator, and $\ll$ is the bitwise SHIFT LEFT operator.

The content of the StreamHash S-BOX computed using the above formula is:

```
760ffb63 74ca107c 8ee6f577 54fd217b 5ca789f2 b5d27f6b 25c2a86f 3624a6c5
89f20430 ca107c01 88978567 32a1f12b 87eabbfe 62ab0ed7 acaa62ab c5073876
4f9274ca ff7d1382 78c1ddc9 4716ff7d 61d82dfa c01fcb59 e1e0a047 43648cf0
e52a95ad 005248d4 cd803aa2 4eb679af a41dde9c e23b49a4 01094072 bff4bac0
66d3a9b7 b72054fd 4486dc93 4568f726 7f6b0536 5e9d753f 6e4568f7 6db34bcc
95ad1834 a86f06a5 9635d9e5 2332a1f1 670aa371 dfef61d8 b4c6c731 1fcb5915
a789f204 8db4c6c7 68f72623 c7312ec3 2a95ad18 d0609096 d27f6b05 506cb89a
24a6c507 c1ddc912 7abdcd80 5a4698e2 721ee9eb b34bcc27 b89a37b2 585e9d75
107c0109 8bceec83 0aa3712c 803aa21a b679af1b b9db9f6e e4aebe5a f8e1e0a0
fb630052 4698e23b 2c42f6d6 eb3c6db3 6f06a529 138211e3 cb59152f 8acf5f84
fc55ed53 37b23ed1 0ffb6300 b0fc55ed d3a9b720 94e7b0fc 9be8c8b1 c912395b
f577026a bac01fcb 69e4aebe ddc91239 42f6d64a 06a5294c 77026a58 f37e8acf
d15170d0 0b9edfef 8191acaa 38760ffb 3aa21a43 8211e34d 312ec333 c4889785
db9f6e45 28ee99f9 e6f57702 d5b5d27f 55ed5350 1ee9eb3c 56b9db9f 3f25c2a8
b23ed151 85670aa3 7c010940 738f738f 5f844f92 6a585e9d a6c50738 198ee6f5
```

Input Data	0
Input Length	0
Comment	empty input
After Update	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Updated with Remaining Bits	4e79f498 d48d0683 e4beabea b7b044c7 ffd6a5b0 e36ba4f4 0d2c3196 ab5199e0
Updated with State Vector	f1f1ebf9 429fcf18 11102b1b 05c02d40 09af7c1a d0e80146 f21c3410 45ded5f2
After State Vector Diffusion	bb432c74 daf0840b a50aac0f ba86e77f 2514b275 b59c0e37 5e392ed5 64caea41
Copied to Returned Vector	bb 43 2c 74 da f0 84 0b a5 0a ac 0f ba 86 e7 7f 25 14 b2 75 b5 9c 0e 37 5e 39 2e d5 64 ca ea 41
After Returned Vector Diffusion	ab fe 2a 9e 78 68 ec f7 9c a6 52 61 1b a1 88 07 2c 40 f2 67 1c b8 c6 fd 5b 94 c2 97 fb c5 af f0

Table 1: Numerical Example 1

Input Data	0
Input Length	1
Comment	single bit of 0
After Update	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Updated with Remaining Bits	f79e57c9 e6beda6c d162bae5 41d26fcd c3c9cc9a 8af7bdc3 fe5dc62a f0e397eb
Updated with State Vector	7271d0db 5199b4da d6a94c0d e65970d8 f42d3589 b116e769 357c25dd 527247dc
After State Vector Diffusion	a21c0b04 ab8f2483 2b021a1a 2b83524b a140029f 3d4f127f 352ca7c8 db4b0a56
Copied to Returned Vector	a2 1c 0b 04 ab 8f 24 83 2b 02 1a 1a 2b 83 52 4b a1 40 02 9f 3d 4f 12 7f 35 2c a7 c8 db 4b 0a 56
After Returned Vector Diffusion	f1 be c9 cd 78 07 2b ae d9 db f5 0f 3a bd 0f 5a fb 3b 3d dc 19 68 7a f9 2e 5a 01 c9 a4 ef f9 4f

Table 2: Numerical Example 2

Input Data	128
Input Length	1
Comment	single bit of 1
After Update	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Updated with Remaining Bits	e6170e86 6d5145f2 596d09d9 9093fac4 176e61e1 07800e4b f995b95e 9bf1e395
Updated with State Vector	25ca4417 2ce6a63c cb659b58 07cd098a da14c117 a78169c7 292e2a8f ffd0a902
After State Vector Diffusion	7040175a 14b6db5c 7bc11d15 08aeee03 95950f3d c9f4ea8a cefcaa79 a47cf80c
Copied to Returned Vector	70 40 17 5a 14 b6 db 5c 7b c1 1d 15 08 ae ee 03 95 95 0f 3d c9 f4 ea 8a ce fc aa 79 a4 7c f8 0c
After Returned Vector Diffusion	5f b0 c7 21 35 eb c6 22 9d 5e 7b 90 98 46 34 37 cc 61 70 ad 76 6a 54 de ac a8 52 cb 6f eb e3 ef

Table 3: Numerical Example 3

Input Data	0
Input Length	8
Comment	single byte of 0
After Update	760ffb63 74ca107c 8ee6f577 54fd217b 5ca789f2 b5d27f6b 25c2a86f 3624a6c5
Updated with Remaining Bits	e0c7aede c2727b90 c944bf71 aa6ed8db d69af83d c5489655 5c5ce1f6 77c2bb74
Updated with State Vector	2939ccbd 5220c577 26108b64 7bdadad6 5fe028d1 07bf6392 1c1eed70 98968f61
After State Vector Diffusion	25d9e724 ba8fbf7e 3a2a3da9 d70ab442 f1114c1a 8ffd68c3 cfedf7ad 49798839
Copied to Returned Vector	25 d9 e7 24 ba 8f bf 7e 3a 2a 3d a9 d7 0a b4 42 f1 11 4c 1a 8f fd 68 c3 cf ed f7 ad 49 79 88 39
After Returned Vector Diffusion	d7 fe e5 09 c3 52 11 8f c9 f3 30 d9 b0 ba 6e b0 a1 b2 fe 18 a7 a4 0c cf 9e 8b 82 2f 78 f1 79 b2

Table 4: Numerical Example 4

Input Data	1
Input Length	8
Comment	single byte of 1
After Update	74ca107c 760ffb63 54fd217b 8ee6f577 b5d27f6b 5ca789f2 3624a6c5 25c2a86f
Updated with Remaining Bits	b353893c 0d8acc21 4c211147 99b6a36b d77a0905 729f7541 17536bbe 21b2a7f2
Updated with State Vector	782a3e2a 786de365 577672d1 523f4366 299f0f74 8c7928d6 2afdfa46 78c36b48
After State Vector Diffusion	8903b6e7 8c6041da 3158ab91 709eed0e 73d215c5 c76b4e8c 966891a9 598d4a64
Copied to Returned Vector	89 03 b6 e7 8c 60 41 da 31 58 ab 91 70 9e ed 0e 73 d2 15 c5 c7 6b 4e 8c 96 68 91 a9 59 8d 4a 64
After Returned Vector Diffusion	7e 8c 42 29 b5 15 56 30 61 b9 64 f5 65 03 f0 fe 71 43 58 1d e4 4f 9d 29 bf 27 b8 61 ba 47 91 f5

Table 5: Numerical Example 5

Input Data	"The quick brown fox jumps over the lazy dog"
Input Length	344
Comment	sample text
After Update	f4a4a20a 5e5c6741 dccab008 e792950c c3e0090e d4a83c26 b26c1b0c b5fd523f
Updated with Remaining Bits	62dacc51 08b88473 4ab4de53 37bae3cc 559e6755 2412b46b 24127557 c9050a4d
Updated with State Vector	b8398a6a d0cc1df7 90579868 e589d6d7 8f7d216e 233265fa fef1336c d44f0947
After State Vector Diffusion	53114925 56da61df 364db12e 14a8181d 8166b81a 9fbbf71f 6e990898 c24cf5cc
Copied to Returned Vector	53 11 49 25 56 da 61 df 36 4d b1 2e 14 a8 18 1d 81 66 b8 1a 9f bb f7 1f 6e 99 08 98 c2 4c f5 cc
After Returned Vector Diffusion	87 64 ad d2 28 02 63 42 78 c5 76 a4 b8 60 78 95 16 7c 34 4e ed a8 9f be 2c c5 cd 65 27 73 68 34

Table 6: Numerical Example 6

```
e34d65bc 152f4eb6 395b57da 2054fd21 9274ca10 a04716ff 0ed70df3 03d5b5d2
da7abdcd eabbfe0c 16ff7d13 3d8bceec 7e8acf5f 1cc48897 79af1b44 648cf017
de9c1cc4 d64a5ca7 d70df37e 4bcc273d a21a4364 a5294c5d 5248d419 8f738f73
5170d060 bbfe0c81 cf5f844f 1b4486dc 86dc9322 35d9e52a 70d06090 9c1cc488
aebe5a46 183428ee 53506cb8 d82dfa14 49a41dde 026a585e a1f12b0b b156b9db
41f8e1e0 f7262332 bdcd803a 9785670a 98e23b49 c2a86f06 6b053624 f6d64a5c
753f25c2 c33366d3 0c8191ac 91acaa62 fe0c8191 d9e52a95 99f969e4 2f4eb679
932294e7 149be8c8 6cb89a37 e9eb3c6d 294c5d8d 217b03d5 59152f4e 3366d3a9
ed53506c e8c8b156 3008bff4 f01787ea 11e34d65 5b57da7a f969e4ae f2043008
08bff4ba 4d65bc78 9d753f25 c6c7312e 1dde9c1c 053624a6 4c5d8db4 5d8db4c6
fa149be8 bc78c1dd 844f9274 f4bac01f 3c6db34b 57da7abd cc273d8b 0df37e8a
3ed15170 9a37b23e 7b03d5b5 2ec33366 63005248 fd217b03 712c42f6 aa62ab0e
9edfef61 60909635 12395b57 c8b156b9 af1b4486 65bc78c1 3b49a41d f12b0b9e
8341f8e1 ec8341f8 be5a4698 7d138211 ee99f969 909635d9 48d4198e dc932294
2dfa149b 0940721e 8cf01787 40721ee9 273d8bce e7b0fc55 ad183428 2b0b9edf
1a43648c 262332a1 4a5ca789 ab0ed70d 043008bf d4198ee6 a3712c42 9f6e4568
ceec8341 3428ee99 ef61d82d 0738760f 2294e7b0 a9b72054 1787eabb e0a04716
```

1.7 Design Rationale

1.7.1 Weaknesses of Commonly Used Constructions

Almost every popular hash function construction performs its function in three basic steps:

- Decomposing of the input stream to blocks;
- Performing compression function on each block; and
- Combining these blocks to produce message digest.

Merkle-Damgard construction is the most commonly used example of the above. The third of the above steps is inherently vulnerable to attacks exploiting differential effects between subsequent input stream bytes.

Most common hash functions avoid utilizing S-BOX tables to reduce the cost of low-end hardware implementations. On the other hand cryptoanalytic efforts of recent years tend to suggest, that alternative techniques intended to provide cryptographic nonlinearity are significantly less secure.

1.7.2 Principles of StreamHash Construction

The StreamHash structure is very different to commonly used constructions. Instead of achieving the avalache effect with multiple rounds, it directly updates the state vector on each input byte.

StreamHash is also based on a well-studied Constraint Satisfaction Problem (CSP).

The reversible transformation performed during the finalization is designed to improve statistical properties of the output. The only security property provided by this transformation is to prevent length-extension attacks.

2 Tunable Security Parameters

No tunable security parameters are defined for the StreamHash algorithm. A weakened version of the algorithm for the cryptoanalysis may be produced by changing the digest size to a lower value.

3 Estimated Computational Efficiency and Memory Requirements

Table 7 shows the approximate number of cycles needed to compute StreamHash.

StreamHash Digest Size	CPU Cycles per Byte
224	20
256	23
384	33
512	43

Table 7: StreamHash Clock Cycles

The algorithm requires no additional setup, thus there is no setup overhead.

Relative throughput of StreamHash is illustrated by figure 1.

The algorithm is also very efficient on 8-bit platforms. It only requires approximately four times more CPU instructions compared to 32-bit architectures. The memory footprint for performance-optimized 8-bit implementations is lower than for 32-bit implementations – 256 instead of 1024 bytes.

4 Known Answer Tests (KATs) and Monte Carlo Tests (MCTs)

Known Answer Tests (KATs) and Monte Carlo Tests (MCTs) values are submitted electronically.

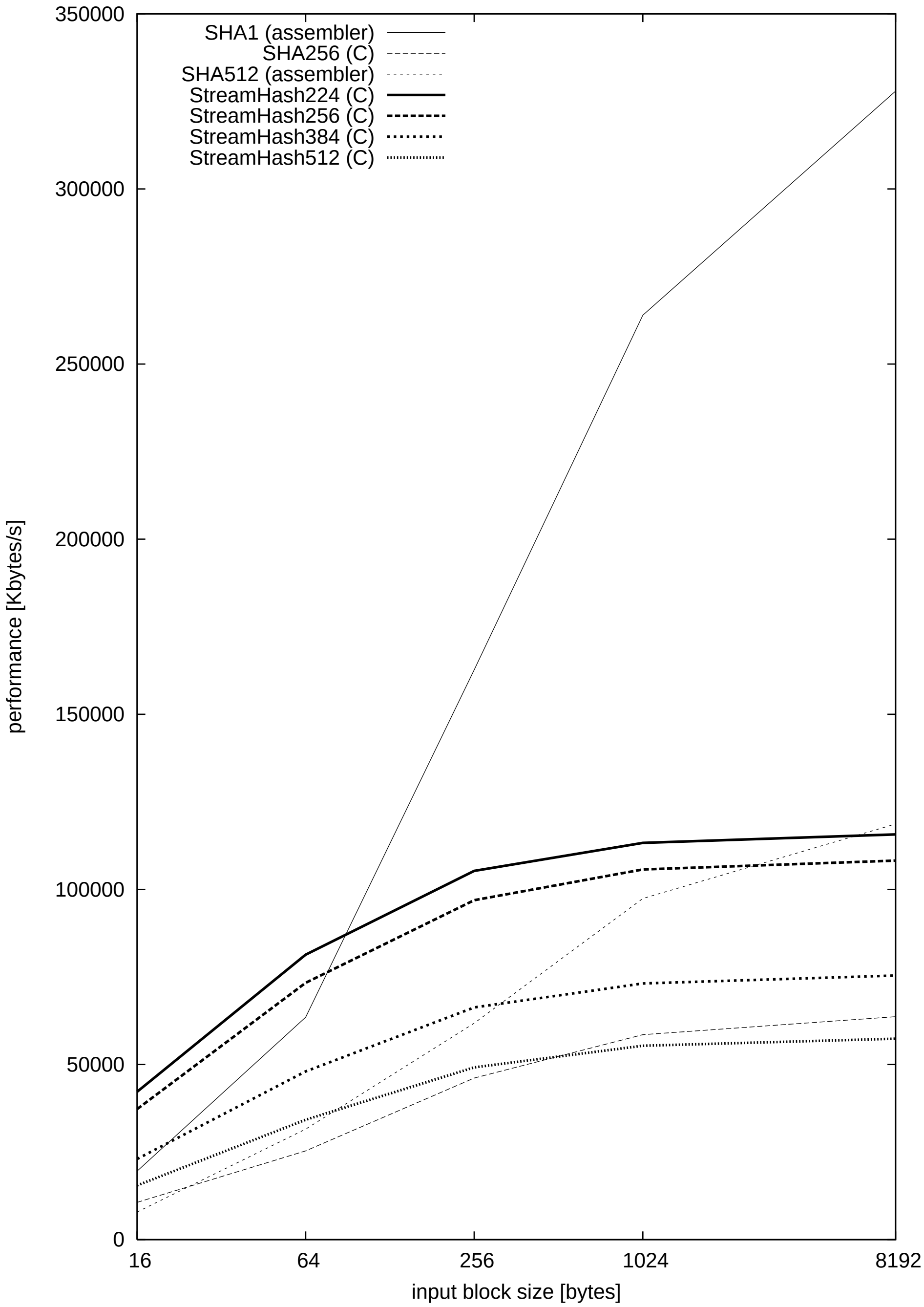

Figure 1: Hash Function Performance Comparison

5 Expected Algorithm Strength

The strength of the StreamHash algorithm is based on the Constraint Satisfaction Problem. No solution was identified to the specified problem more effective than a brute force search, exponential to the digest size.

StreamHash can be utilized for HMAC and PRFs. For HMAC the recommended construction is to prepend the message with the secret key. StreamHash may be utilized for PRFs by computing the aforementioned HMAC values on a counter. It is required to only use first half of the message digest bits to assure resistance against distinguishing attacks.

StreamHash security properties also apply to any subset of its input bits. The next section provide statements on Streamhash resistance against known attacks.

6 Resistance to Known Attacks

6.1 Collision Finding Attack

Collision finding without birthday attack implies solving the Constraint Satisfaction Problem. No collision finding attack was identified against Streamhash more effective than the birthday attack.

6.2 First Preimage Finding Attack

Each unknown byte of the preimage results in nonlinear transformation of the whole state vector. No method easier than exhaustive search was found for solving the system of non-linear equations in order to find the preimage.

6.3 Second Preimage Finding Attack

Second preimage finding attack implies solving the Constraint Satisfaction Problem.

6.4 Length Extension Attack

The finalization phase updated state vector with data derived from the state. This is supposed to provide more effective resistance against length-extension attacks, than appending the length utilized in hash functions based on the Merkle-Damgard construction.

6.5 Multicollision Attack

StreamHash is not affected with multicollision attack, as it is not based on the Merkle-Damgard construction.

6.6 Constraint Satisfaction Problem Attacks

Common algorithms for solving Constraint Satisfaction Problem include backtracking, constraint propagation, and local search. The StreamHash algorithm is build, so that common algorithms to solve Constraint Satisfaction Problems cannot be applied. This property is ensured by the clear separation of the constraints. Solving a subset of all constraints does not make solving remaining constraints easier.

6.7 Constants and Tables

No constants or tables are used in the algorithm, other than the S-BOX based on well-studied AES S-BOX.

6.8 Prior Cryptoanalysis

No prior third party work describing or analyzing the security of the submitted algorithm is known to the submitter.

7 Advantages and Limitations

7.1 Advantages

- Very high cryptographic strength;
- Clear and easy to analyze design;
- No performance penalty on high-endian systems;
- High efficiency for 8-bit implementations (only four times more CPU instructions are required compared to 32-bit architectures);
- Highly parallelizable for hardware implementations, allowing to process data at single clock cycle per input byte;
- Low finalization lattency, property important for real-time (e.g. multimedia) applications;

- Minimal code size, property important for embedded systems;
- Minimal state size, property important for embedded systems;
- High throughput for short messages; and
- Simple transformation to other message digest sizes (any multiple of 32 bits).

7.2 Limitations

- Relatively expensive (in the number of gates) to implement hardware, as StreamHash is an S-BOX is used;
- Some constant data needed for S-BOX (1024 bytes on 32-bit platforms and 256 bytes on 8-bit platforms), unless S-BOX values are computed on the fly; and
- Only 70% more throughput of StreamHash-256 compared to SHA-256 for bulk data processing.