The SANDstorm Hash

Mark Torgerson and Richard Schroeppel
Tim Draelos, Nathan Dautenhahn, Sean Malone, Andrea Walker,
Michael Collins, Hilarie Orman*

Cryptography and Information Systems Surety Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0785

*Purple Streak, Inc.
500 S. Maple Dr.
Woodland Hills, UT 84653

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Table of Contents

List of Figures

1. Introduction

The SANDstorm hash family is designed for maximal cryptographic strength and high speed on most common architectures. Other design features are:

- Speed improvements strongly correlated with the number of processing elements in parallel and/or pipelined architectures.
- Compression function with a novel structure that avoids known weaknesses in older hashes.
- Prevention of length extension attacks.
- Large internal state. This increases the resistance to several kinds of attacks involving large messages or multi-collisions.
- Friendliness to multi-algorithm usage. SANDstorm uses the SHA family constants so that implementations that must support both the SANDstorm family and the SHA family need store only one set of algorithm-specific constants.
- Reuse of the SHA strategy for obtaining 224 bit hashes. The SANDstorm-224 function is the same as for SANDstorm-256 except that different initialization variables are used. The same strategy is used for SANDstorm-384 and SANDstorm-512.
- Compatibility with the NIST standard for HMAC and randomized hashing schemes. This will allow "plug and play" with many of the data formatting mechanisms and program wrappers currently in use.

The SANDstorm family achieves a great deal of mixing while performing on par with the SHA family of algorithms on the 32 bit architectures that we tested. Significant speed gains are realized on 64 bit architectures. For either architecture, a few lines of assembly code realize further gains.

Parallel implementations of two separate parts of SANDstorm account for its high speed potential. The compression function is up to 10 times faster on parallel architectures. Similarly, the tree-based mode operation can be up to 1000 times faster.

The SANDstorm family is API-compatible with the SHA family. This makes SANDstorm suitable for the applications specified by the following publications:

- FIPS 186-2, Digital Signature Standard;
- FIPS 198, The Keyed-Hash Message Authentication Code (HMAC);
- SP 800-56A, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography; and
- SP 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators (DRBGs).

SANDstorm has a design that foils collision attacks, preimage attacks, and second preimage attacks. We know of no design weaknesses that would render it less secure than the theoretic bounds for the output sizes.

Design Overview

SANDstorm has 4 hashes in its family:

- SANDstorm-256 and -224 Hash operates on 512 bit blocks, and the algorithm definition is based on operations using 64 bit words.
- SANDstorm-512 and -384 Hash operates on 1024 bit blocks, and the algorithm is based on operations using 128 bit words.

There are four main components in the design

- Padding
- Mode
- Chaining
- Compression

Each of these will be explained in subsequent sections.

All the hashes use a mode that is a modified and truncated tree with a finalization step. Within the mode, the Merkle-Damgard chaining has a novel structure that is notably different from the MD5 and SHA chaining. The compression function is particularly efficient.

Notation and Conventions

Because the SANDstorm family has four hashes and two block sizes, we will use a shorthand notation in discussion of the algorithms. For example, “a 512(1024) bit block” means “a 512 bit block for SANDstorm-224 and SANDstorm-256 (or a 1024-bit block for SANDstorm-384 and SANDstorm-512)”. A similar interpretation applies to “a 64(128) bit word”. Section 4 below discusses other notations that apply to the two hash sets.

The following section on notation and conventions was taken almost verbatim from various portions of FIPS PUB 180-3 dated October 2008; it can be found at: http://csrc.nist.gov/publications/PubsFIPS.html

We have used the same notational conventions as in the FIPS documents. There are a few rearrangements and deletions of the FIPS text and also a few additions.

Symbols

The following symbols are used in the SANDstorm algorithm specifications, and each operates on w-bit words:

&	Bitwise AND operation
$\oplus$	Bitwise XOR (“exclusive-OR”) operation
$\neg$	Bitwise complement operation
+	Addition modulo 2^w
*	Multiplication modulo 2^w
|	Concatenation Operation

<< Left-shift operation, where $x << n$ is obtained by discarding the left-most n bits of the word x and then padding the result with n zeroes on the right.

>> Right-shift operation, where $x >> n$ is obtained by discarding the rightmost n bits of the word x and then padding the result with n zeroes on the left.

$ROTL^n(x)$
The rotate left (circular left shift) operation where x is a w-bit word and n is an integer with $0 \leq n < w$, is defined by $ROTL^n(x) = (x << n) \oplus (x >> w - n)$. Thus, $ROTL^n(x)$ is equivalent to a circular shift (rotation) of x by n positions to the left.

Bit Strings and Integers

The following terminology related to bit strings and integers will be used.

1. A hex digit is an element of the set {0, 1,…, 9, a,…, f}. A hex digit is the representation of a 4-bit string. For example, the hex digit “7” represents the 4-bit string “0111”, and the hex digit “a” represents the 4-bit string “1010”.

2. A word is a w-bit string that may be represented as a sequence of hex digits. To convert a word to hex digits, each 4-bit string is converted to its hex digit equivalent, as described in (1) above. For example, the 32-bit string

1010 0001 0000 0011 1111 1110 0010 0011

can be expressed as “a103fe23”, and the 64-bit string

1010 0001 0000 0011 1111 1110 0010 0011
0011 0010 1110 1111 0011 0000 0001 1010

can be expressed as “a103fe2332ef301a”.

3. An integer between 0 and 2^{32}-1 inclusive may be represented as a 32-bit word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. For example, the integer $291 = 2^8 + 2^5 + 2^1 + 2^0 = 256+32+2+1$ is represented by the hex word 00000123.

The same holds true for an integer between 0 and 2^{64}-1 inclusive, which may be represented as a 64-bit word. Similarly for other sized integers as well.

A SANDstorm implementation usually operates on 64(128) bit words, but occasionally the words are broken into half-size pieces. For example, for a 64 bit word, if Z is an integer, $0 \leq Z < 2^{64}$, then $Z = 2^{32}X + Y$, where $0 \leq X < 2^{32}$ and $0 \leq Y < 2^{32}$.

4. For the SANDstorm family of hash algorithms, the size of the message block depends on the algorithm.
a) For SANDstorm-224 and SANDstorm-256, each message block has 512 bits, which are represented as a sequence of eight 64-bit words. Thus, a 512 bit data block D may be represented as $D = (d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7)$ where the d_i are 64 bit words. In the

description below, 256 bit quantities are passed from one functional block to another and are represented as four 64 bit words, e.g. $E = (e_0, e_1, e_2, e_3)$.
b) For SANDstorm-384 and SANDstorm-512, each message block has 1024 bits, which are represented as a sequence of eight 128 bit words. Similarly, a 1024 bit data block can be represented as $D = (d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7)$ where the d_i are 128 bits in length.

2. The SANDstorm Padding and Message Length

The message padding for SANDstorm is simple. A message is padded by appending a 1-bit and then appending 0-bits until the result length is a multiple of the block length.

Let B be the block length and η be the length in bits (before padding) of the message to be hashed. If η is a multiple of B, then a block consisting of a 1-bit followed by B-1 0-bits will be appended to the message.

The message length η must be passed into the mode for use in the finishing step.

All members of the SANDstorm hash family support message lengths of $0 \leq \eta < 2^{128}$.

3. The SANDstorm Mode

Many hash functions, such as SHA, use a mode based on the Merkle-Damgard chaining construction. The chaining is deliberately sequential, and it does not benefit from parallel or pipelined implementations. Tree-based block combining is highly parallelizable, but its drawback is that the message size determines the depth of the tree and the amount of storage. This variability makes the implementation of tree based hashing challenging. Also, latency issues may arise with a full tree-based approach.

The SANDstorm mode is a truncated tree with bounded depth and storage. It permits efficient parallel implementations. The amount of intermediate storage is at most 10 blocks. For very long messages the algorithm could take advantage of up to 1000 processing elements with near linear speed-up.

Each level of the tree uses the SANDstorm chaining (described below) to build the tree. The mode also provides for a finishing step as well as an “early-out” option to speed-up small message processing.

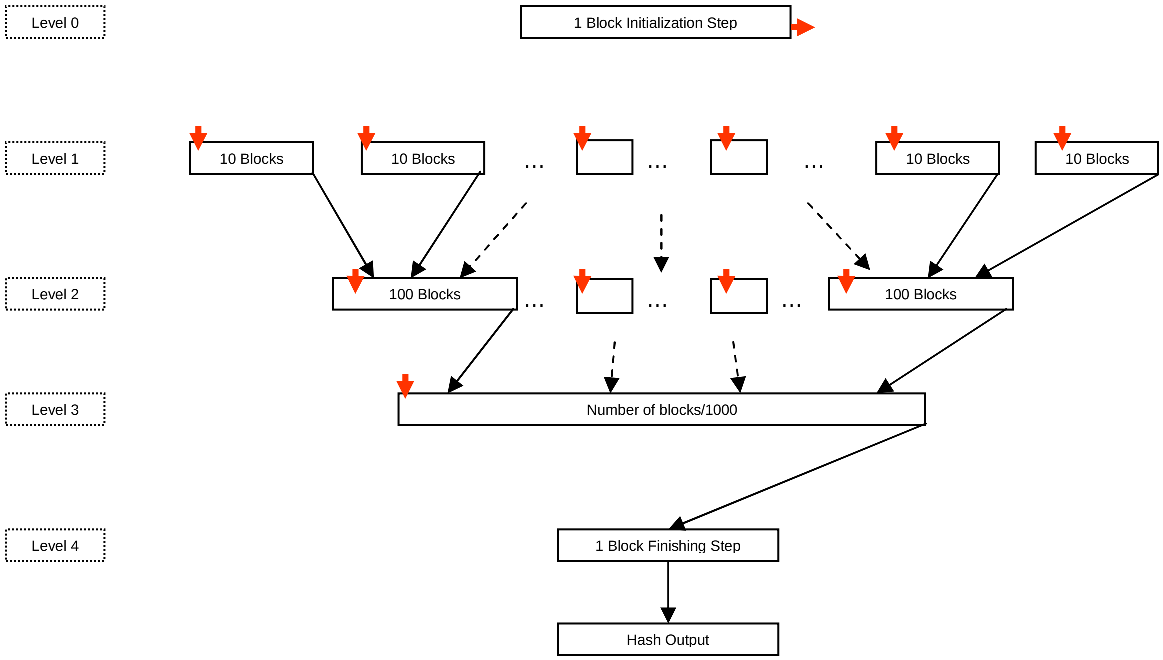

Figure 1: SANDstorm Mode. The red arrow always denotes the output of Level 0. The rectangles with solid outlines represent functions.

SANDstorm Mode Description

In the following discussion, the reader should assume that the input message has been padded and parsed into an integral number of 512(1024)-bit blocks.

The SANDstorm mode has five different levels; the number of levels actually used varies with the message length. A level is a function that takes one or more blocks (or superblocks, defined below) as input and produces a list of blocks as output.

- Level 0 and Level 4 are completed no matter the length of the input message, and each takes a single block as input. The output from Level 0 is used in Levels 1, 2, and 3.
- Use of Levels 1, 2, and 3 depends on the length of the message. The following conditions define the "early out" mechanism:
    - If the input message has only one block, skip Levels 1, 2, and 3
    - If the output of Level 1 has only block, skip Levels 2 and 3
    - If the output of Level 2 has only one block, skip Level 3

  NB, if an application can guarantee that its messages will always be less than 11 blocks in length, its implementation of SANDstorm need not include code for Level 2 or Level 3. Similarly, Level 3 can be eliminated if all messages are less than 1001 blocks in length.
- Level 4 is a finishing step.

Superblocks. Levels 1, 2 and 3 logically operate on block lists of length 10, 100, or unlimited, respectively. We call these "superblocks". If the input to a chaining function is a quantity N consisting of *n* ordered blocks $N_1, \ldots, N_n$, and if the superblock size parameter for that level is *k*, then we define the superblock T_i as a list

$$T_i = \{ N_{k(i-1)+j} \} \quad 1 \le i \le n/k,\ 1 \le j < k$$

For all blocks except possibly the last, the superblock consists of exactly *k* of the N_i blocks. For the last superblock, if *n* is not a multiple of *k*, then j runs from 1 to *(n mod k)*.

Within a level, the chaining function operates on the input blocks (or superblocks). The chaining function computes 4 state variables for each block and carries these values forward for the next block. After the last block is processed, the state variables are used to compute the output value for the level.

Chaining Function. Within the chaining function, there are 4 internal state variables that are the intermediate values of the generalized Merkle-Damgard chain. For each block position *i* within a super block and for $1 \le j \le 4$ we have chain(j, i) as the chaining value. Below we define initialization constants and the so for $1 \le j \le 4$ we have c_j = chain(j, 0). Finally if the superblock is k long we have the output of the superblock processing is the four chaining values chain(j, k) for each $1 \le j \le 4$. This notation assumes that the superblock in question is known. In reality each superblock is a function of level and position. To more fully show the superblock input we set, for a given superblock, T_i, we set, for $1 \le j \le 4$, chn(j, Ti) to be the output of the superblock. Finally, the output of the superblock is processed to be fed into the next level of the tree.

$$\mathrm{Chn}(T_{i,} \mathrm{IV}) \rightarrow \ (\mathrm{chn}\ (1, T_i) \oplus \ \mathrm{chn}\ (3, T_i) \mid \mathrm{chn}\ (2, T_i) \oplus \ \mathrm{chn}\ (4, T_i))$$

IV are the initialization constants c_j mentioned above. The IV is a function of level and the input message block M_0. The level initialization values are described in the "Initialization Constants" section and Figure 5. The level transition function Chn converts 4 256(512) bit values into a single data block 512(1024) bits in length.

Levels. Up to five levels of processing are used, depending on the length of the message. Longer messages use more levels. Levels 1, 2, and 3 can begin their processing as soon as one superblock is available.

Let M = $M_0, M_1, \ldots, M_m$ be the *m*+1 blocks of the input message after padding and parsing. Each M_i is 512(1024) bits.

■ If m = 0, then the flow of control uses only two levels. Level 0 processes M_0 and passes the output to the finishing step, Level 4. This bypass of Levels 1, 2, and 3 is the most efficient case of "early out".

■ If m > 0, then Level 0 and Level 1 together partially compress M, creating an n-block quantity N = $\{ N_{i=1,n} \}$ where the N_i are 512(1024) bits in length. The superblock size for level 1 is 10; the superblocks for Level 1 are derived from M. If n is not an integer multiple of 10, then N_n is produced by SANDstorm chaining the last (*n* modulo 10) blocks of M.

Each N_i is the result of applying SANDstorm chaining ("Chn") to superblock T_i .
The output of Chn is a block formed by the concatenation of two bitstrings:

$$N_i = (\text{chn}\ (1, T_i) \oplus \text{chn}\ (3, T_i) \mid \text{chn}\ (2, T_i) \oplus \text{chn}\ (4, T_i))$$

■ If $n = 1$, then N_1 is fed directly into the finishing step of Level 4.

■ If $n>1$, then the blocklist N is partially compressed in Level 2 to produce $P = \{ P_{i=1,p} \}$ where again the P_i are 512(1024) bits in length. The superblock size parameter for level 2 is 100. Thus, 100 blocks of N are used to produce one superblock of P. If n is not an integer multiple of 100, then P_p is produced by SANDstorm chaining the last (*n* modulo 100) blocks of N. Here the superblocks are composed of the elements of N.

$$P_i = (\text{chn}\ (1, T_i) \oplus \text{chn}\ (3, T_i) \mid \text{chn}\ (2, T_i) \oplus \text{chn}\ (4, T_i))$$

■ If $p = 1$, then P_1 is fed into the finishing step of Level 4. Otherwise, the message P is fed into Level 3. This level does not use superblock grouping. It uses the chaining function on each ordinary input block in turn. The output of the level is the single block formed by the operation of Chn as above. That output is the input to the Level 4 finishing step.

Message Digest. For SANDstorm hash-224, the final message digest is the XOR of all four 256 bit string outputs of Level 4, with the leftmost 224 bits retained.

For SANDstorm hash-256, the final message digest is the XOR of all four 256 bit string outputs of Level 4.

For SANDstorm hash-384, the final message digest is the XOR of all four 512 bit string outputs of Level 4, with the leftmost 384 bits retained.

For SANDstorm hash-512 the final message digest is the XOR of all four 512 bit string outputs of the Level 4.

SANDstorm Mode Performance

Levels 1 and 2 can benefit from parallel processing architectures. The levels have Merkle-Damgard chaining within the superblocks of size 10 and 100, respectively, but the superblocks themselves can be processed independently. For large messages, this allows a speed up to a factor of 1000 through mode parallelization. Level 3 is processed sequentially.

There is a good deal of flexibility in the construction, and larger messages can take advantage of even more parallel computation. For example, if a message is larger than 12 blocks, there is an opportunity to process two Level 1 superblocks of size 10, independently, which would approximately halve the computation time. For messages greater than 1002 blocks in length, implementers can take advantage of the parallelization in Level 2, with consequent speed-ups approaching a factor of 1000.

On the other extreme, if one does not have the resources to exploit the parallelizability of the SANDstorm mode, then one pays a penalty of having to process the second and third levels of the tree. That then implies a slowdown of a factor of (1+1/10+1/(1000)) over straight sequential processing of the message blocks. That is less than an 11% slowdown for long messages.

Because of the "early out" mechanism for short messages, the processing cost of the levels of the tree, including the impact of the finishing step, is mitigated for shorter messages. A message of length $0 \le \eta < 512(1024)$ bits will pad out to exactly one block. After Level 0 processing, the output will pass directly to Level 4. Only two compressions are needed to process a message of that size. A raw input message that is between $1 \le k \le 11$ blocks long would require $k+1$ compressions including the finishing step.

The second level of the tree is not invoked unless the raw input message is at least 11 blocks in length, requiring 14 compressions. Similarly, Level 3 is not invoked until the raw input message is at least 1002 blocks long.

There is some latency associated with the last block to be processed. When the final message block is received, that block must be processed as it propagates through the levels of the tree. The latency depends on the length of the input message and ranges from two to four compression operations. For messages less than twelve blocks in length the final message block must be run through two compression functions. For messages of length greater than 1001 blocks, four levels must be traversed.

The SANDstorm mode can benefit from precomputation. A prerequisite for precomputation is a constant initial raw message block M_0. The superblocks, especially in Level 1, may be computed independently of the other superblocks. This means that in cases where large messages need to be hashed many times with only small changes between hashings, then much of the hashing work associated with the unchanged portion of the message may be computed and stored. Only the change-affected superblocks in Level 1 and Level 2 need to be recomputed. Of course this may require additional storage. If the places of change are known and fixed in particular superblocks on Level 1, then most of the rest of the message may be processed down to Level 3. This would require the storage of a new message of size roughly m/1000 blocks in length and would see a speed up of nearly a 1000 over simple serial processing. Further, the Level 3 computation could process up to the changed position, reducing the amount of needed storage and computation.

Choice of Superblock Size

The design uses architectural choices that are the result of trade-offs between performance and resources. We want adopters of the hash to see performance benefits commensurate with the resources they can afford. Our choices reflect a reasonable compromise between speed on commodity hardware, speed on specialized parallel hardware, and usability on low-end embedded processors. The design choices are reflected in:

- Number of levels in the tree. Each level uses intermediate storage; too many levels will be a burden on machines with limited storage. The latency is directly related to the number of levels.
- The superblock sizes for Levels 1 and 2. The product of the superblock sizes determines the maximum advantage obtained with parallel processing. The individual superblock sizes determine the minimum processing delay for a message.

4. The SANDstorm Chaining

In the usual iterative Merkle-Damgard construction, the chaining variable is a function of the previous chaining value and the i-th message block, i.e., $h_i = H(h_{i-1},M_i)$. Almost all implementations use XOR for combining the two inputs to H, such as setting $h_i = H(h_{i-1} \oplus M_i)$. This construction is antithetical to parallelization and/or pipelining and is why we chose a more general iterative form.

Another of our design goals was to have a larger amount of internal state than is in the SHA family in order to add resistance to various types of attacks that exploit the size of the internal state, such as, long messages, multi-collisions, and herding, etc,. However there is a direct correlation between performance and the amount of state that is operated on during the course of the compression function.

Our construction uses an iteration on 5 variables:

$$h_i = \{ H(a_{i-1}, b_{i-1}, c_{i-1}, d_{i-1}, N_i), h_A(N_i), h_B(N_i), h_C(N_i), h_D(N_i) \}$$
$$\text{where } a_{i-1} = h_A(N_{i-1}), b_i = h_B(N_{i-1}), c_{i-1} = h_C(N_{i-1}), d_{i-1} = h_D(N_{i-1})$$

The functions H, h_A, h_B, h_C, h_D, are the same for each level. The number of iterations of H is the number of rounds. There are always five rounds.

This iterative form allows us to parallelize within the chaining of the superblocks and to carry forward more internal state while retaining efficiency.

SANDstorm Compression Description

Suppose that $M = M_0,M_1,\ldots,M_m$ is the message to be hashed. Then M is operated on in the SANDstorm truncated tree mode in levels to produce successively more compressed data. The data will be operated on in superblocks of a given size as explained in the section describing the SANDstorm mode. Here we describe the data flows at the superblock level and show how to pass data from one level of the tree to the next.

Level 0
The input to Level 0 is the first 512(1024) bits of the padded and parsed message.

The compression function H has 5 rounds R_i, i=0,1,2,3,4. Each of those rounds takes input from a bank of pre-defined constants $A_{r,i}$ (defined in the compression function definitions below),and from the message schedule. The input constants C_0, C_1, C_2, C_3, and C_4 for each hash in the SANDstorm family are defined below.

The output of Level 0 is the four 256(512)-bit strings S_1, S_2, S_3, and S_4. These four values are subsequently used as inputs for each superblock compression on levels 1, 2, and 3.

If the input message is one block long, S_1, S_2, S_3, and S_4, are combined into an input block for Level 4:

$$\text{Level 4 Input} = (S_1 \oplus S_3 \mid S_2 \oplus S_4)$$

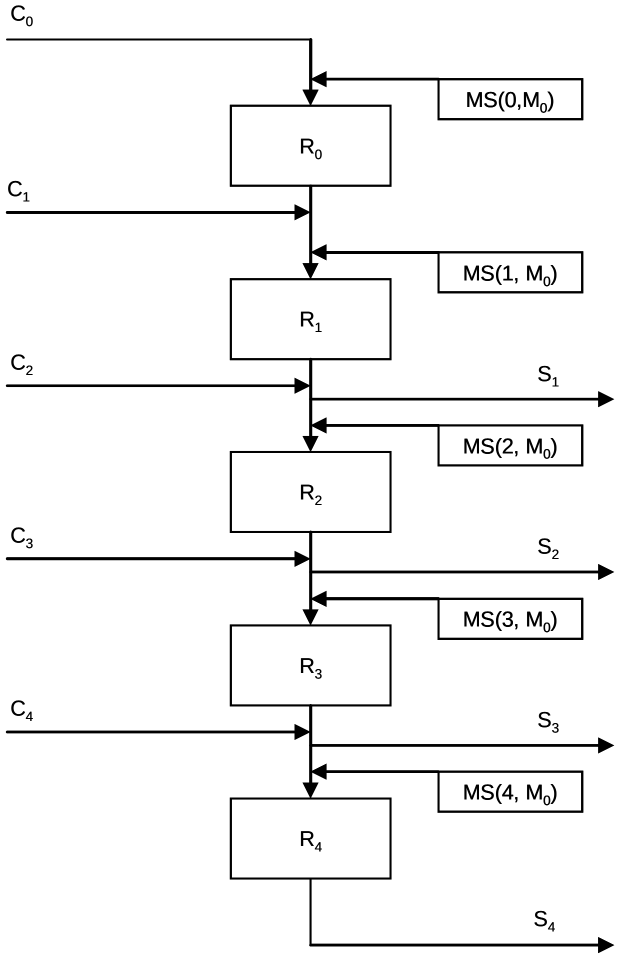

Figure 2: Level 0

In figure 2,

- $R_{i=0,1,2,3,4}$ are the round functions.
- $MS(r, M_0)$ is the message schedule with a 256(512) bit output, which is a function of the round number and data block M_0.

R and MS will be explained in detail in the compression function section.

The input into R_0 is $C_0 \oplus MS(0,M_0)$
For $0<i\leq 4$ the input to R_i is $R_{i-1} \oplus C_i \oplus MS(i,M_0)$

For $1 \leq i \leq 3$ the output $S_i = R_i \oplus C_{i+1}$
The output $S_4 = R_4$

Note that in the description of Level 0 we have used M_0 as an input for the message schedule. Level 0 always operates only on the first message block M_0. No other level does this. For the other levels, we use D in place of M because the blocks may be derived from computations on input message blocks.

Levels 1, 2, and 3
Levels 1, 2, and 3 chain multiple blocks of data together. These blocks are the sub-blocks of a superblock. Given a sequence of j 512(1024) bit data blocks $D_1, D_2, \ldots, D_j$ we process these j blocks sequentially. Each data block is processed as in Level 0 except that the four 256(512)-bit output strings act as the input values c_i for the next block. For instance, the output of round three in the compression function for block i will be part of the input for round three in block i+1.

The following graphic illustrates the state transition from one block to another.

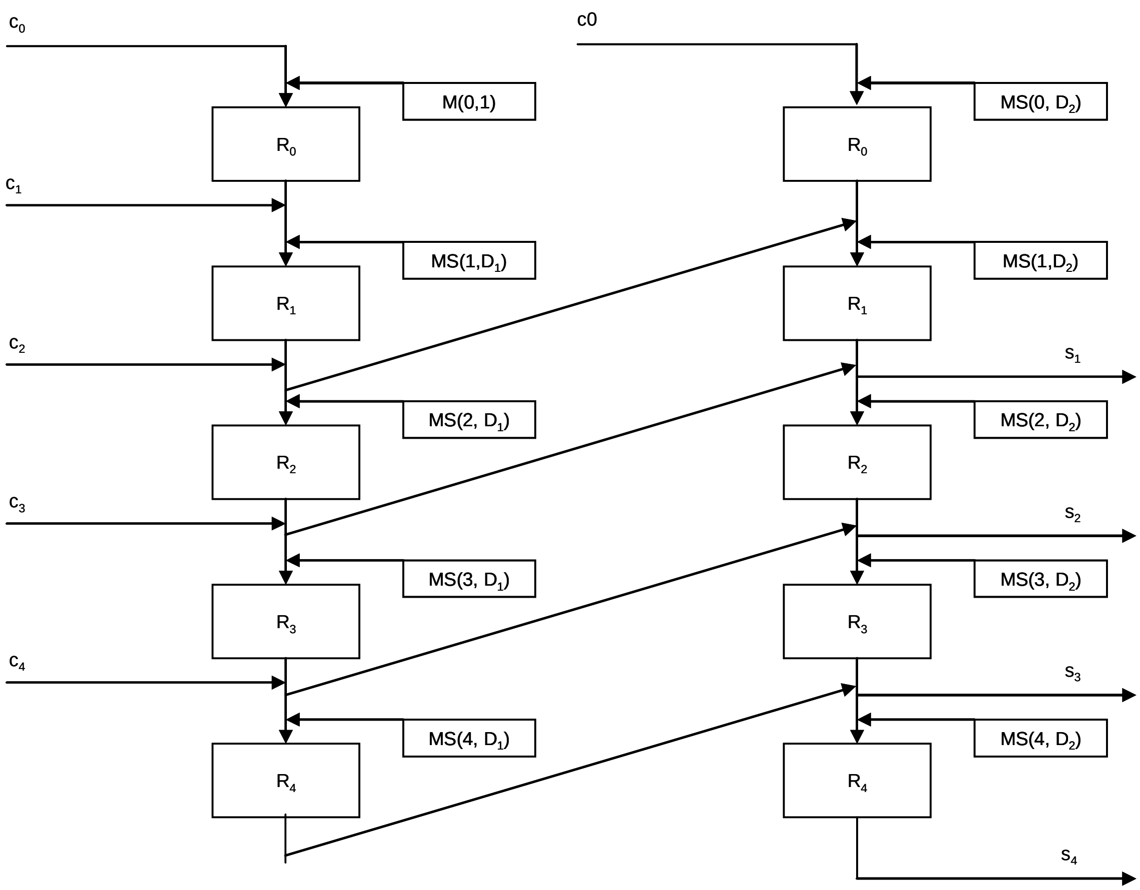

Figure 3: Two Block Superblock

The illustration uses a superblock of size 2. Each arrow represents 256(512) bits. When an arrow joins another, the combiner operation is XOR.

To extend this method to larger superblocks, one continues the construction for R_{i+1} on block D_j by computing the XOR of

- the output state R_i on block D_j
- the chaining value from R_{i+1} from the previous block D_{j-1}
  (forward the result to the combiner for the previous round R_i for the next block D_{j+1})
- the message schedule words MS(i+1, D_j) for round i+1.

In other words, for $1\leq i \leq 4$ and $1 \leq k \leq j$ the input into round i at block position k is:
$R(i-1,k) \oplus chain(i,k-1) \oplus M(i,D_k)$

We also have for $1 \leq i \leq 3$ and $1 \leq k \leq j$ $chain(i,k) = R(i-1,k) \oplus chain(i+1,k-1)$
For $1 \leq k \leq j$ $chain(4,k) = R(4,k)$.

This arrangement has two interesting facets. (1) It's pipeline friendly. (2) The output of round 4 affects the input of round 4 for the next block, and the input of round 3 for the following blocks, and the input of rounds 2 and 1 of later blocks.

Note that superblocks of many sizes are possible. In Level 1, the size is bounded by 10. In Level 2 the size is bounded by 100, and in Level 3 by roughly m/1000.

Levels 1 and 2 have initialization values c_0, c_1, c_2, c_3, and c_4. Each of these values is a function of the Level 0 input constant C_i, the Level 0 output string S_i , and the superblock number *i*. Level 3 is similar, but the initialization does not depend on the superblock number. The formulas for the c_i are given below.

At the end of the superblock compression the four 256(512)-bit output values of rounds 1 through 4 (s_1, s_2, s_3, and s_4) are combined to produce the 512(1024) bit input value for the next level of the tree: $(s_1 \oplus s_3 \mid s_2 \oplus s_4)$

Level 4

The finishing step, Level 4, is similar to Level 0. It operates on a single block of data of length 512(1024) bits.

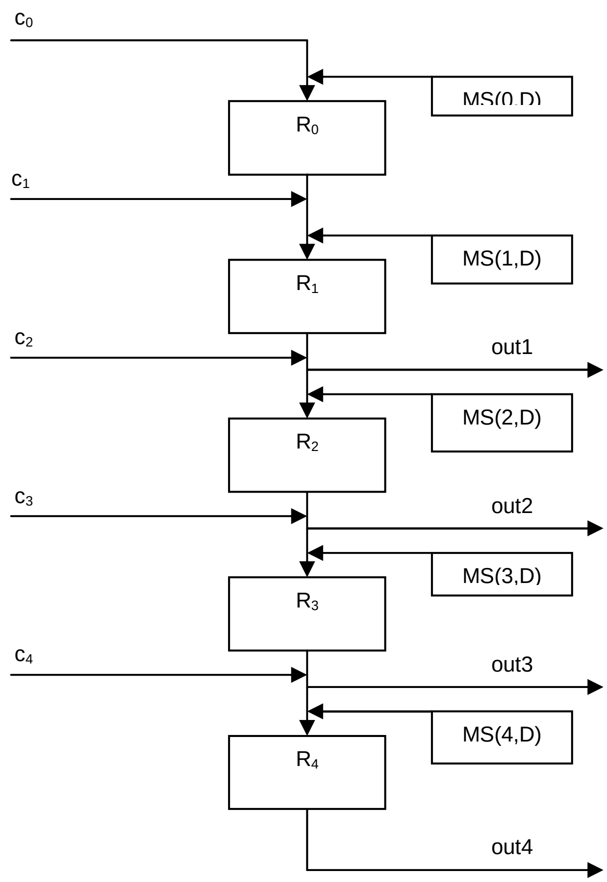

Figure 4: Level 4

The input values are defined below, and they are functions of the prepadded message bit length, η. The input message for Level 4 is the single block output $D = (s_5,s_6)$ from one of the previous 4 levels, depending on the message length as explained in the SANDstorm mode section. The final message digest of the message M is the XOR of the outputs from rounds 1 through 4:

$$\text{SANDstormHash}(M) = \text{out1} \oplus \text{out2} \oplus \text{out3} \oplus \text{out4}$$

Initialization Constants

For each level of the SANDstorm mode, the values c_0, c_1, …, c_4 are a function of C_0, C_1, …, C_4 and S_1, S_2, S_3, and S_4. For Level 1 and Level 2 the constants are also a function of the block position. Since there is only one superblock on Level 3, that level does not include a superblock number.

In the following, we assume the input is a padded and parsed message $M = M_0,M_1,\ldots,M_m$.

For SANDstorm-256 and -224, Level 1 processes 10 blocks at a time. For superblock i, represent the index as i as a 128 bit number, and $\alpha_i= (i,i)$ is 256 bits in length. For Level 1, let k be the smallest integer greater than m/10, then the counter has values $1 \le i \le k$.

SANDstorm-512 and -384 also have a message length bounded by 2^{128}. The 128 bit counter i is viewed as a 256 bit integer so that $\alpha_i = (i,i)$ is 512 bits in length.

Level 2 processes the outputs of Level 1 in superblocks of size 100. Each data block in Level 2 is the result of SANDstorm chaining of 10 input messages blocks. So each superblock in Level 2 is a function of message blocks $M_{(i-1)*1000+1}$, $M_{(i-1)*1000+2}$,…, and M_{i*1000}. Set $\beta_i = (i, i)$. For Level 2, let k be the smallest integer greater than m/1000, then the counter is in the range $1 \leq i \leq k$.

As above, for SANDstorm-512 and -384 we have that $\beta_i = (i,i)$ is 512 bits in length.

For SANDstorm-256 and -224 let c and d be defined as the first 64 bits of the fractional part of the fifth root of 5 and 7 respectively. Represented in hexadecimal notation:
c = `6135f68d4c0cbb6f`
d = `79cc45195cf5b7a4`
Let $\beta = (0, 0, 0, c)$ and $\delta = (0, 0, 0, d)$ be two 256 bit strings.

For SANDstorm-512 and -384 let c and d be defined as the first 128 bits of the fractional part of the fifth root of 5 and 7 respectively. Represented in hexadecimal notation:
c = `6135f68d4c0cbb6fb43b47a245778989`
d = `79cc45195cf5b7a4aec4e7496801dbb9`
Let $\beta = (0, 0, 0, c)$ and $\delta = (0, 0, 0, d)$ be two 512 bit strings.

The message length, before padding, η, in bits, is included in Level 4. Let $\varepsilon = (\neg\eta,\eta)$, which is viewed as a 256(512) bit string.

	Level 0	Level 1	Level 2	Level 3	Level 4
c_0	C_0	$C_0 \oplus S_4 \oplus \alpha_i$	$C_0 \oplus S_4 \oplus \beta \oplus \beta_i$	$C_0 \oplus S_4 \oplus \delta$	$C_0 \oplus \delta \oplus \varepsilon$
c_1	C_1	$C_1 \oplus S_1 \oplus \alpha_i$	$C_1 \oplus S_1 \oplus \beta \oplus \beta_i$	$C_1 \oplus S_1 \oplus \delta$	$C_1 \oplus \delta \oplus \varepsilon$
c_2	C_2	$C_2 \oplus S_2 \oplus \alpha_i$	$C_2 \oplus S_2 \oplus \beta \oplus \beta_i$	$C_2 \oplus S_2 \oplus \delta$	$C_2 \oplus \delta \oplus \varepsilon$
c_3	C_3	$C_3 \oplus S_3 \oplus \alpha_i$	$C_3 \oplus S_3 \oplus \beta \oplus \beta_i$	$C_3 \oplus S_3 \oplus \delta$	$C_3 \oplus \delta \oplus \varepsilon$
c_4	C_4	$C_4 \oplus S_4 \oplus \alpha_i$	$C_4 \oplus S_4 \oplus \beta \oplus \beta_i$	$C_4 \oplus S_4 \oplus \delta$	$C_4 \oplus \delta \oplus \varepsilon$

Figure 5: Initialization Values

$C_0, C_1, \ldots, C_4$ each are 256(512) bit words comprised of the SHA initialization constants. We denote those constants as $H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7$.

$C_0 = (H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7)$
$C_1 = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_0)$

$C_2 = (H_2, H_3, H_4, H_5, H_6, H_7, H_0, H_1)$
$C_3 = (H_3, H_4, H_5, H_6, H_7, H_0, H_1, H_2)$
$C_4 = (H_4, H_5, H_6, H_7, H_0, H_1, H_2, H_3)$

The following are the initial values for SHA-224, which are given in hexadecimal. These are used to create the C_i for SANDstorm-224. These are the low 32 bits of the constants for SHA-384 (see below).

H_0 = c1059ed8
H_1 = 367cd507
H_2 = 3070dd17
H_3 = f70e5939
H_4 = ffc00b31
H_5 = 68581511
H_6 = 64f98fa7
H_7 = befa4fa4

The following are the initial values for SHA256, which are given in hexadecimal, and which are obtained by taking the first 32 bits of the fractional part of the square root of the first eight prime numbers. These are used to create the C_i for SANDstorm-256.

H_0 = 6a09e667
H_1 = bb67ae85
H_2 = 3c6ef372
H_3 = a54ff53a
H_4 = 510e527f
H_5 = 9b05688c
H_6 = 1f83d9ab
H_7 = 5be0cd19

The following are the initial values for SHA-384, which are given in hexadecimal. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers. These are used to create the C_i for SANDstorm-384.

H_0 = cbbb9d5dc1059ed8
H_1 = 629a292a367cd507
H_2 = 9159015a3070dd17
H_3 = 152fecd8f70e5939
H_4 = 67332667ffc00b31
H_5 = 8eb44a8768581511
H_6 = db0c2e0d64f98fa7
H_7 = 47b5481dbefa4fa4

The following are the initial values for SHA-512, which are given in hexadecimal. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers. These are used to create the C_i for SANDstorm-512

H_0 = `6a09e667f3bcc908`
H_1 = `bb67ae8584caa73b`
H_2 = `3c6ef372fe94f82b`
H_3 = `a54ff53a5f1d36f1`
H_4 = `510e527fade682d1`
H_5 = `9b05688c2b3e6c1f`
H_6 = `1f83d9abfb41bd6b`
H_7 = `5be0cd19137e2179`

Parameters and Performance

In the SANDstorm chaining, if each message or data block is processed sequentially, as may happen in software, then the throughput associated with SANDstorm chaining in the superblocks is comparable to a standard Merkle-Damgard hash. However, if one has sufficient resources available, and can take advantage of the connections between rounds, pipelining can speed-up the computation. For example, each round might be a separate stage of the pipeline. That pipeline will emit a result as often as the slowest round completes.

For the SANDstorm compression function, about 60% of the work is done in the message schedule, which can be parallelized and pipelined, so if the resources are available, a factor of 7 increase in speed over sequential processing is possible. This does not include any gains one might see by including additional resources to speed up the round function itself.

5. The SANDstorm Compression Function

The SANDstorm compression function H has two main components: the round function and the message schedule.

- The SANDstorm round function R is a one-to-one function of 256(512) bits organized as four 64(128) bit words. R has two different sections, one more algebraic and the other more logical in nature.

- The SANDstorm message schedule operates on eight 64(128) bit words in a one-to-one fashion.

SANDstorm-256 and -224 Compression Function Description

<u>General Functions</u>

Let $Z=X*2^{32}+Y$ be a 64 bit word, where Y and Z are 32 bits each.
Define functions:

- $ROTL^n(Z)$ is a left rotation of Z by n positions
- $F(Z) = X^2+Y^2$ modulo 2^{64}
- $G(Z) = X^2+Y^2+ROTL^{32}((X+a)(Y+b))$ modulo 2^{64}
  The additions X+a and Y+b are taken modulo 2^{32} before the product (X+a)(Y+b) is computed. The product is viewed as a 64 bit quantity and so the rotation is a swap of the high and low order halves. The constants a and b are defined below.
- $Ch(A,B,C) = (A\&B)\oplus(\neg A\&C)$
- SB(Z) = Z except that low order byte, z, of Z is replaced with the AES sbox(z)

The constants a and b are defined as the first 32 bits of the fractional part of the fifth root of 2 and 3 respectively, with the high and low bits forced to one. Represented in hexadecimal notation, they are:

```
a = a611186b
b = bee8390d
```

BitMix Function

The BitMix function operates at the bit level on four 64 bit state words to produce four 64 bit state words. There are four 64 bit constants that select separate bit positions of a 64 bit word. Given in hexadecimal, these are:
J_8 = 8888888888888888
J_4 = 4444444444444444
J_2 = 2222222222222222
J_1 = 1111111111111111

If A, B, C, D are all 64 bits in length, then (A', B', C', D') = BitMix(A, B, C, D), where
$A' = (J_8 \& A) \oplus (J_4 \& B) \oplus (J_2 \& C) \oplus (J_1 \& D)$
$B' = (J_8 \& B) \oplus (J_4 \& C) \oplus (J_2 \& D) \oplus (J_1 \& A)$
$C' = (J_8 \& C) \oplus (J_4 \& D) \oplus (J_2 \& A) \oplus (J_1 \& B)$
$D' = (J_8 \& D) \oplus (J_4 \& A) \oplus (J_2 \& B) \oplus (J_1 \& C)$

The BitMix function can be viewed as a permutation of the bits in each column of the 64-by-4 (128-by-4) bit matrix formed by A, B, C, D.

Round Function

The round function R consists of two parts. The first is a mixing of the four state words with, primarily an integer multiplication. The second is a bit mixing that helps destroy the algebraic properties associated with the multiplication.

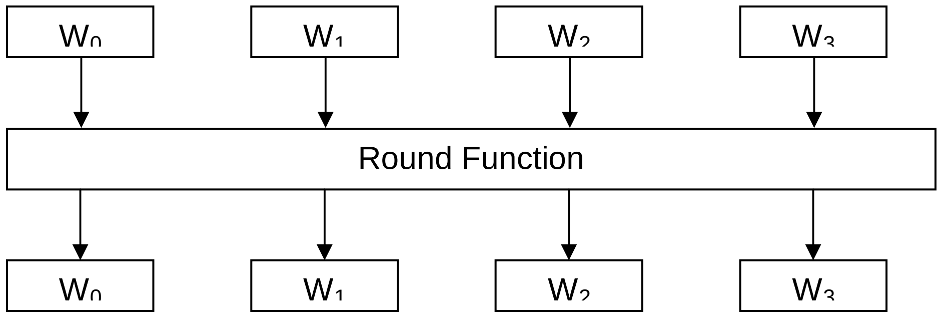

Figure 6: Round Function

Round Function

The SANDstorm-224 and -256 round function is a one-to-one function operating on four 64 bit words and producing four 64 bit words.

For SANDstorm-224 and SANDstorm-256, for each round do the following:
For i from 1 to 3
    Set $W_i = \text{ROTL}^{25}(\text{SB}([W_i + F(W_{i-1}) + \text{Ch}(W_{i-1},W_{i-2},W_{i-3}) + A(r,i)] \text{ modulo } 2^{64}))$
Set $(W_0, W_1, W_2, W_3) = \text{BitMix}(W_0, W_1, W_2, W_3)$

The A(r,i) are round constants defined below, where r is the round number and i is the word position number. In the For loop, the subscripts are taken modulo 4, and the computations of W_i are assumed to be iterative, so that as each value is updated the new value is used to update subsequent values. As mentioned earlier, the BitMix function can operate on the words in parallel.

The effect of the tunable security parameter is to repeat Round 4 a specified number of extra times. The default is to execute Round 4 exactly once with no extra repetitions. The repetitions re-execute the round function formula above with the round variable fixed at r=4, followed by the BitMix. The parameter should be an even number between 0 and 20. The default value is 0.

Message Schedule

The message schedule receives a 512 bit block of data viewed as eight 64 bit words. Given input data block $D = (d_0, d_1, \ldots, d_7)$ the eight words are expanded to a total of 33 64 bit words.

For i from 8 to 32
    Set $d_i = \text{ROTL}^{27} (\text{SB}([d_{i-8} + G(d_{i-1})+\text{Ch}(d_{i-1},d_{i-2},d_{i-3}) + d_{i-4} + B_i] \text{ modulo } 2^{64}))$

In the SANDstorm chaining description we used the notation MS(r,D) to denote the contribution from the message schedule for round r as operated on data block D.
$\text{MS}(0,D) =$
    $\text{BitMix}(\text{ROTL}^{19} (d_0) \oplus d_4, \text{ROTL}^{19} (d_1) \oplus d_5, \text{ROTL}^{19} (d_2) \oplus d_6, \text{ROTL}^{19} (d_3) \oplus d_7)$
$\text{MS}(1,D) = (d_{14}, d_{15}, d_{16}, d_{17})$

$MS(2,D) = (d_{19}, d_{20}, d_{21}, d_{22})$
$MS(3,D) = (d_{24}, d_{25}, d_{26}, d_{27})$
$MS(4,D) = (d_{29}, d_{30}, d_{31}, d_{32})$

Constants

SHA-224 and SHA-256 use the same sequence of sixty-four constant 32-bit words. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. SANDstorm uses the first 50 of the SHA constants, $K_0, K_1, \ldots, K_{49}$. In hexadecimal, these 50 constant words are (from left to right)

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08
```

The constants B_i in the message schedule are 64 bits in length and are formed by concatenating the SHA-256 constants, that is:
For i from 8 to 32
    Set $j = i-8$
    Set $B_i = K_{2j}*2^{32} + K_{2*j+1}$

There are 20 constants A(r,i). They are equal to the B_i but are in reverse order, that is:
For $0 \le r \le 4$ and $0 \le i \le 3$
    Set $A(r,i) = B_{32-(4*r+i)}$

SANDstorm-512 and -384 Compression Function Description

The SANDstorm-512 and -384 round function is a one-to-one function operating on four 128 bit words and producing four 128 bit words.

General Functions

Let $Z = X*2^{64}+Y$ be a 128 bit word, where Y and Z are 64 bits each.
Define functions:

- $ROTL^n(Z)$ is a left rotation of Z by n positions
- $F(Z) = X^2+Y^2$ modulo 2^{128}
- $G(Z) = [X^2+Y^2+ROTL^{64}((X+a)(Y+b))]$ modulo 2^{128}
    - The additions Y+a and Z+b are taken modulo 2^{64} before the product (X+a)(Y+b) is computed. The product is viewed as a 128 bit quantity and so the rotation is a swap of the high and low order halves. The constants a and b are defined below.
- $Ch(A,B,C) = (A\&B)\oplus(\neg A\&C)$
- SB(Z) = Z except that low order byte, z, of Z is replaced with the AES sbox(z)

The constants a and b are defined as the first 64 bits of the fractional part of the fifth root of 2 and 3 respectively, with the high and low bits forced to one. Represented in hexadecimal they are:

```
a = a611186bae67496b
b = bee8390d43955aed
```

BitMix Function

The bit mix function operates at the bit level on four 128 bit state words to produce four 128 bit state words. There are four 128 bit constants that select separate bit positions of a 128 bit word. Given in hexadecimal, these are:

J_8 = 88888888888888888888888888888888
J_4 = 44444444444444444444444444444444
J_2 = 22222222222222222222222222222222
J_1 = 11111111111111111111111111111111

If A, B, C, D are all 128 bits in length, then (A', B', C', D')=BitMix(A, B, C, D), where

$A' = (J_8 \,\&\, A) \oplus (J_4 \,\&\, B) \oplus (J_2 \,\&\, C) \oplus (J_1 \,\&\, D)$
$B' = (J_8 \,\&\, B) \oplus (J_4 \,\&\, C) \oplus (J_2 \,\&\, D) \oplus (J_1 \,\&\, A)$
$C' = (J_8 \,\&\, C) \oplus (J_4 \,\&\, D) \oplus (J_2 \,\&\, A) \oplus (J_1 \,\&\, B)$
$D' = (J_8 \,\&\, D) \oplus (J_4 \,\&\, A) \oplus (J_2 \,\&\, B) \oplus (J_1 \,\&\, C)$

Round Function

For each round do the following:
For i from 0 to 3
    Set $W_i = \mathrm{ROTL}^{57}(\mathrm{SB}\,([W_i + F(W_{i-1}) + \mathrm{Ch}(W_{i-1},W_{i-2},W_{i-3}) + A(r,i)] \text{ modulo } 2^{128}))$
Set $(W_0, W_1, W_2, W_3) = \mathrm{BitMix}(W_0, W_1, W_2, W_3)$

The A(r,i) are round constants defined below, where r is the round number and i is the word position number. In the For loop the subscripts are taken modulo 4 and the computations of W_i are assumed to be iterative, so that as each value is updated the new value is used to update subsequent values. As mentioned earlier, the BitMix function operates on the words in parallel.

The effect of the tunable security parameter is to repeat Round 4 a specified number of extra times. The default is to execute Round 4 exactly once with no extra repetitions. The repetitions re-execute the round function formula above with the round variable fixed at r=4, followed by the BitMix.

Message Schedule

The message schedule receives a 1024 bit block of data viewed as eight 128 bit words. Given input data block D = (d0, d1, …, d7) the eight words are expanded to a total of 33 128 bit words.

For i from 8 to 32

Set d_i = $ROTL^{59}$ (SB([d_{i-8} + G(d_{i-1})+Ch(d_{i-1},d_{i-2},d_{i-3}) + d_{i-4} +B_i] modulo 2^{128}))

In the SANDstorm chaining description we used the notation MS(r,D) to denote the contribution from the message schedule for round r as operated on data block D.
MS(0,D) =
BitMix($ROTL^{37}$ (d_0) ⊕ d_4, $ROTL^{37}$ (d_1) ⊕ d_5, $ROTL^{37}$ (d_2) ⊕ d_6, $ROTL^{37}$ (d_3) ⊕ d_7)
MS(1,D) = (d_{14}, d_{15}, d_{16}, d_{17})
MS(2,D) = (d_{19}, d_{20}, d_{21}, d_{22})
MS(3,D) = (d_{24}, d_{25}, d_{27}, d_{27})
MS(4,D) = (d_{29}, d_{30}, d_{31}, d_{32})

Constants

SHA-384 and SHA-512 use the same sequence of eighty constant 64-bit words. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. SANDstorm-512 and -384 will use 50 of those constants, K_0, K_1, …, K_{49}. In hexadecimal, these constant words are (from left to right)

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53
```

The constants B_i in the message schedule are 128 bits in length and are formed by concatenating 50 of the the SHA-512 constants, that is:

For i from 8 to 32
Set j = i-8
Set B_i = K_{2j}*2^{64} + K_{2*j+1}

There are 20 constants A(r,i). They are equal to the B_i but are in reverse order, that is:
For 0 ≤ r ≤ 4 and 0 ≤ i ≤ 4
Set A(r,i) = $B_{32-(4*r+i)}$

SANDstorm Compression Function Performance

SANDstorm relies on multiplication as a primary mixing agent. On most modern computers this operation is efficient. Since multiplication inherently does a very good job of mixing, we don’t need a great number of rounds to accomplish our design goals. Thus we have a small number of rounds that heavily mix the data. The smaller number of rounds and relative speed of the individual operations allows for an efficient design.

As mentioned above, the message schedule performs much of the mixing, and it does not depend on the state variables. That means a large fraction of the work in the compression function may be accomplished in parallel and/or pipelined processing elements.

By having one round output feed forward as input in the same round of the next block, the latency associated with the typical Merkle-Damgard construction is reduced to something more manageable. One must account for the latency of a single round.

There are implementations where one could expand the SHA initialization constants to create the SANDstorm initialization constants each time a new superblock is created. This would take a tiny fraction of the time required to compress one block of data. However, the more reasonable approach in most software applications is to expand the initialization values and fix them as part of the source code. In this case, there is no time required before Level 0 processing on the first block can commence.

NIST has asked that submissions demonstrate that the algorithm uses all table values. The construction of SANDstorm ensures that every block will exercise every value in every table, with the exception of the sbox.

The sbox table is used 45 times during one block compression (including computation of the message schedule). A hash of 50 blocks will do 2250 sbox look-ups, averaging 8.8 touches per table entry. The expected number of untouched entries is about $256*e^{-8.8} = 0.03$. It is probable that every table entry is touched at least once.

6. Cutdown and Extension Alternatives

NIST has asked submitters to provide appropriate cutdown functions for analysis and to provide a method to extend the algorithm to have more strength if deemed necessary.

The simplest cutdown method would be to chop off some number of rounds starting with round 4. Round 0 is different than the rest and does not make for a good chopping point. The smallest reasonable place to cut to is right after Round 1. By chopping to Round 1 there would still be a chaining value from previous blocks.

Of course, if the algorithm is cut down to Round 1, the output size is only 256 bits. This small amount of chaining state necessarily would lose the resistance to multicollisions, herding, etc.

Chopping other rounds, provided the appropriate chain forward values are kept, would keep in the spirit of the algorithm. We also assume that if rounds are cut out of one portion of the algorithm (Level in the mode) that all other compressions, no matter what level in the mode, will be cut in a similar fashion.

We don’t feel that there is a security risk in changing the predefined superblock sizes. However, we do not recommend using variable block sizes

We don’t believe it is necessary to increase the strength of the algorithm, but since NIST requested it, we provide a couple of possibilities.

Option 1:
We may post process the chaining values as they are produced. The amount of state being carried from one block position to the next is significant, so additional processing of one or more of the chaining values may give the desired enhancement. In particular, the information as output from Round 4 is at least the size of the final message digest. Therefore processing that information further may be a simple and straightforward way to implement and provide whatever security enhancement is needed.

For instance, Round 4 may be repeated a specified number of times. That is, we take the output of Round 4 and run it through the round function again. This does not include additional stepping of the message schedule, it just requires repetition of the For loop and the BitMix. Since all outputs are combined and eventually fed into Level 4, the finishing step provides additional strength. We have implemented this option as a #define in the reference implementation. The parameter should be an even number between 0 and 20; the default value is 0.

Option 2:
Increase the number of rounds. Extend the message schedule, computing five steps per round and using four. The round functions can be added on, with the chaining values linking the last four rounds in the pattern above. There are a number of unused SHA constants. The B_i and the A(r,i) can be defined appropriately.

This option makes most sense if changes are completed before widespread implementation. The issue is that the connections between rounds would have to be changed and the constants reworked to line up with the right rounds. This option may not be attractive if implemented after the fact.

7. Design Choices

In this section we discuss and elaborate on several design features of the SANDstorm family.

- The method of padding for the SANDstorm family differs from that of the SHA family by not appending the length. The SANDstorm family uses a 128 bit length counter. Appending the length counter would often add an extra block to the message. Our mode includes superblock numbers in each superblock, and the finishing step includes the bit length. The finishing step prevents length extension attacks, and so SANDstorm’s padding is suitable and only rarely requires an extra block.

- The SANDstorm constants were chosen to be those either used by SHA-256 or derived in the same fashion. This saves memory space in situations where both SHA and SANDstorm might be simultaneously implemented. We also favor the SHA constants because and they are public and the generation method is well-known.
- The constants a and b in the message schedule have high and low bits set to 1. This ensures a broader set of values as output of the multiplication.
- Squaring and multiplication operations are the workhorses for effecting the bit mixing in the F and G functions. We use a function that was inspired by 1's complement squaring but turns out to be better for mixing. The function
  $Z^2 \bmod 2^{64}-1$ has the property that each bit of output is a function of each bit of input, and thus this is a fairly good mixing agent. The downside to it is that low Hamming weight words stay low Hamming weight. In particular, a one bit change in a low weight input has limited effect on the output. The cross term in the function $G(Z)= X^2+Y^2+ROTL^{32}((X+a)(Y+b))$ is designed to force a small change in low weight inputs to be noticeable. The $ROTL^{32}$ is just an efficient approximation of what happens to the cross terms of the square mod $2^{64}-1$.
- The function $F(Z) = X^2+Y^2$ in the round function is an efficient version of G(Z). It doesn't mix as well, but we wanted to make sure there was sufficient difference between the message schedule operations and the round function operations.
- There is an application of the AES sbox in the low order byte of certain words during the round function and the message schedule. The AES sbox is highly non-linear and provides excellent mixing for the bits that it acts on. The choice to apply the sbox on the low order byte was for efficiency's sake. Our sbox has at position x the value
  $x \oplus sbox_{AES}(x)$
  so that we can, with a single xor, replace x with sbox(x). Indexing by any other byte position would require more operations.
  Further, the application of the sbox is not our primary mixing operation; it is there to defeat differentials in the low order byte position. To propagate small changes an attack would have to repeatedly avoid the low order byte.
  There are 45 applications of the sbox. Each is accompanied by a rotation; there are two different rotation constants.
- The BitMix function was chosen as a method to break up any algebraic dependencies that might appear in the round function and cause further separation between the words in the message schedule.
- The BitMix function depends on each state word within the round. After the BitMix, each data nibble contains information from each word of the round state. Mixing operations in the next round will destroy any correlations that may have existed in the inputs to BitMix.
- The mode was designed specifically so that parallelization of superblock operations would be possible.
- The structure of the chaining values between rounds has two benefits: careful management admits some parallelization and pipelining within the round function; long message attacks are mitigated.

8. Security Discussions

We believe that the SANDstorm family satisfies all of the usual security requirements for a cryptographic hash. That is, for a w = 224, 256, 384, 512 bit message digest sizes the work required for function inversion (preimage) is on the order of 2^w compressions. Similarly, the work to find collisions is on the order of $2^{w/2}$. In addition, the SANDstorm mode and chaining structures increase the work required for long message attacks to equal that of inversions. Removal of long message attacks adds significant resistance to second preimage attacks. Further, SANDstorm carries 4w bits of state from block to block. This means other attacks that randomly exploit the internal state and chaining values will be foiled. Thus, multicollision and herding type attacks will be infeasible.

General Observations

Collisions in the Chaining Values with changes in the message

It is a straightforward exercise to show that if, for any i = 1 to 3, we have that MS(i,D) = MS(i,D') and MS(i+1,D) = MS(i+1,D'), then D = D'. The function in the message schedule was designed to fill the gap in the values pulled out of the schedule.

This means that if the message input (excluding the contribution to round zero) taken in adjacent pairs is the same, i.e. a collision on the message contribution, then the input messages have to be the same. This means that if $D \neq D'$ there must be a difference in at least two non-adjacent contributions from the message schedule.

From this we can show that, given a two strings of data blocks that are identical up to one point, the chaining values cannot collide at the point of difference. In a given superblock, suppose D and D' are at block position j and suppose that the two data strings are equal up to that position. The chaining values coming into position j must be the same.

Now suppose that the chaining values moving into position j+1 are equal. Starting with the last couple of rounds we have, by assumption, that chain(4,j) = chain'(4,j) and that chain(3,j) = chain'(3,j). For the first equality to hold, the inputs to Round 4 must be the same. The inputs are a sum of the chaining variables and the message schedule. For Round 4 we have that chain(3,j) $\oplus$ MS(4, D) = chain'(3,j) $\oplus$ MS(4,D'), and so MS(4,D) = MS(4,D'). Similarly by equating chain(2,j) and chain'(2,j) we determine MS(3,D) = MS(3,D'). From above, this means that D = D'. This means that a change in one data block will be guaranteed to propagate at least into the next block position.

If the changed block happens to be at the end of a superblock, although the chaining values will be different, we would like to know that the resulting data feeding into the next level will be different. We do not have a proof of this.

Message Schedule Security

Given an input data block D = d_0, …, d_7 the message schedule is a powerful mixing operation. From d_8 through d_{32}, each word is progressively less correlated with the message input words. The last word with measurable correlation is d_{12}. Our analysis of the correlation follows.

Each bit of word d_{12} depends on each bit of $d_0, \ldots, d_7$ except d_3. The word d_3 enters d_{11} as a simple sum of the other words, the low byte mapped with the sbox, and rotated. Deltas in d_3 are passed directly into G(Z) in the computation of d_{12}. Empirically, there are a couple of weak bits, namely those d_3 bits that rotate into bit positions 60-63. We ran a series of tests comparing one bit deltas in each of the 512 input bits of the $d_0, \ldots, d_7$. Bit position 36 of d_3 yields noticeable non-uniform statistics in many bit positions of d_{12}. To a much lesser degree, so do positions 35, 34, and 33 of d3.

The rotation value of 27 was chosen so that d_3 deltas in the low order byte are first operated on by the sbox and then rotated into bit positions 28-31. Other rotation amounts where the delta is not operated on by the sbox, but rotates into positions 28-31, will also give non-uniform results for d_{12}. The rotation value of 27 was chosen to make sure that the sbox output sits in the top of the low order half of d_{12}.

Even though there a couple of weak bit positions in d_3 as viewed from d_{12}, the rest of the bit positions of $d_0, \ldots, d_7$ have a fairly uniform affect on d_{12} and there are no weak bits when viewed from d_{13}. As a rough gauge of the mixing ability of the messages schedule, we have that each bit of d_{i+13}, d_{i+14}, d_{i+15}, is a strong function of each bit of $d_i, \ldots, d_{i+7}$. The message schedule steps 25 times and so there are effectively three passes through input data. Each pass results in an excellent mixing of the previous eight words. Single bit differentials of random data do not propagate more than a few steps. Once the delta is operated on by G(X) an avalanche effect occurs.

The SANDstorm message schedule skips the first six values and then outputs four. Each bit of $MS(1,D) = (d_{14}, d_{15}, d_{16}, d_{17})$ is a strong function of each bit of D. Similarly, each bit of $MS(2,D) = (d_{19}, d_{20}, d_{21}, d_{22})$ is a strong function of each bit of $(d_6, \ldots, d_{13})$
$MS(3,D) = (d_{24}, d_{25}, d_{27}, d_{27})$ is a strong function of each bit of $(d_{11}, \ldots, d_{18})$
$MS(4,D) = (d_{29}, d_{30}, d_{31}, d_{32})$ is a strong function of each bit of $(d_{16}, \ldots, d_{23})$

Round Function

The round function is not quite as complex as the message schedule and so does not mix quite as well. However, the multiplications are still very effective. There are fewer mixing steps in the round functions than in the message schedule. However, the BitMix function removes the algebraic structures that may arise in the first part of the round function. Let (W_0, W_1, W_2, W_3) be the inputs to the round function, let (W'_0, W'_1, W'_2, W'_3) be what is produced by the For loop in the round function, and let $(W''_0, W''_1, W''_2, W''_3) = \text{BitMix}(W'_0, W'_1, W'_2, W'_3)$. Although$W'_0$ is not a strong function of all of the W_i, the strength increases with For loop iterations, and at the end, each bit of W'_3 is a strong function of each bit of the input words, W_i. After the BitMix operation, each byte of each of the W''_i has two bits from each of the W'_i, and so we may say that each byte of W''_i is a strong function of each input bit of the W_i. The output of the For loop in the next round turns each bit of its output into a strong function of each input bit of the previous round. This means that the output of Round 4 has seen more than two full mixes of Round 0 inputs and two mixes of Round 1 inputs. Each of the chaining values is a full mix of the data two rounds previous.

Specific Observations

<u>Second Preimage attacks</u>

One method of generating second preimages with very long messages in a typical Merkle-Damgard construction is to create a second message that varies from the original toward the beginning of the message but keeps the rest of the message the same. Past the changed blocks, the message is the same, so if there is ever a collision in the chaining values, the collision will persist and the two messages will collide to create a second preimage.

For a really long message, Level 3 acts like a typical Merkle-Damgard construction so it will be susceptible to the ills of that construction. That is, long message attacks, multicollisions and herding are all possible. However these methods of attack require one to get collisions in the chaining values. SANDstorm's chaining values carry forward at least four times as much state as is in the final hashing value. These state bits are not completely independent, but we will show evidence that our constructions are strong enough to completely foil certain attacks.

When applying the second preimage attack described above to SANDstorm, we suppose that for some very large t two data strings $D = D_1, \ldots, D_t$ and $D' = D'_1, \ldots, D'_t$ that the strings differ at the beginning of the message and agree after position K. Assume that holds for K < j-2. Since the message schedule inputs are the same at position j-1 and j, we have that if we assume that chain(3,j-1) = chain'(3,j-1), then this implies that we necessarily have that chain(4, j-1) = chain'(4, j-1). If we assume further that chain(1, j) = chain'(1, j), it is an easy exercise to show that this forces chain(2, j) = chain'(2,j), chain(3, j) = chain'(3, j), and chain(4, j) = chain'(4, j) to also hold. So we have shown that if we assume that the two conditions

chain(3,j-1) = chain'(3,j-1) AND chain(1, j) = chain'(1, j)

simultaneously hold, then the chaining values collide in the j-th position and the messages D and D' collide.

Now suppose that chain(3,j-1) = chain'(3,j-1) but chain(1, j) ≠ chain'(1, j). Then chain(4,j-1) = chain'(4,j-1) and chain(2, j) ≠ chain'(2, j). Together these imply chain(3, j) ≠ chain'(3, j).

Similarly, suppose that chain(1, j) = chain'(1, j) but chain(3,j-1) ≠ chain'(3,j-1). Then chain(2, j) ≠ chain'(2, j). Both chain(1, j) and chain'(1, j) become inputs for position j+1. Since they are equal and the other inputs are equal too, then the outputs of Round 1 in position j+1 must be equal. However, chain(2, j) ≠ chain'(2, j), and both combine with the outputs of Round 1 to create chain(1, j+1) and chain'(1, j+1) which forces them to be unequal, thus breaking the linking of the Round 1 chaining variables.

These two cases show that if one pair of chaining values are equal but the other pair are unequal, then the pair equality is guaranteed to be destroyed. That in turn means that if chain(3,j-1) = chain'(3,j-1) and chain(1, j) = chain'(1, j), which is enough to force a collision in the chaining values, those two conditions had to be met simultaneously. So with digest size w, the resistance is on the order of 2w-k bits for a message of size 2^k bits. When k=w the success rate is on par with finding a single preimage, which requires work on the order of 2^w operations.

Collisions

The previous section indicates that there is little advantage in using long messages to create collisions. This means that an attack might as well be based on short messages, or on somehow exploiting the level structure.

The size of data being passed from one level to the next is 2w bits, in other words, twice that of the final message digest size. Any randomly generated attack to generate collisions in the superblock output will require work on the order of 2^w operations.

Multicollisions may be constructed by choosing messages that differ by a block but collide, then extending the message with two new blocks to collide starting with the collided chaining variable. In SANDstorm one may attempt this, but one must either force the chaining variables to collide or else force a collision in the level data. Both take work on the order of 2^w. This effectively removes the possibility of success. Herding type attacks are similar and have the same work bounds.

Our expectation is that collisions stemming from manipulation of the mode and chaining constructs will require on the order of 2^w operations. This clearly requires more work than just finishing the hash and finding collisions in the message digest directly.

Length Extension Attacks

The final message digest is related to the chaining values in a complicated way. The finishing step, the padding and the length field in the finishing step of Level 4 effectively remove the possibility of doing length extensions.

9. Application USE, HMAC, etc.

In all ways, the SANDstorm family is designed to be a drop-in replacement for the SHA family, and for each digest size the SANDstorm family will be have strength equal to or greater than the corresponding member of the SHA family, no matter what the application, including any existing application of HMAC, Pseudo Random Functions, and Randomized Hashing.

10. Computational Efficiency

Our computational efficiency estimates are based on the reference platform indicated in the NIST documentation. Our tests were run on

- NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.

- Compiler (Note that the selection of this compiler is for use by NIST in Rounds 1 and 2, and does not constitute a direct or implied endorsement by NIST.): the ANSI C compiler in the Microsoft Visual Studio 2005 Professional Edition.

Due to the method of construction, the timings for SANDstorm-224 and SANDstorm-256 are virtually identical, similarly for SANDstorm-512 and -384.

An optimized version of SHA-1 and SHA-256 were used as reference points of comparison. The NIST api seemed to get in the way and cause our timing routines to give odd results. The timings below bypass the more external functions and focus on the time to complete a single compression function. These do not count the finishing step, which may be amortized away with long messages.

According to the call for proposals not much if any priority in Round 1 of the competition will be given to assembly coded implementations. However, we experience difficulty with the reference compiler during a multiplication of two 32 bit numbers where the 64 bit output was retained. It had a tendency to convert the 32 bit numbers to 64 bit numbers and then do the multiplication. This irritating operation slowed down our implementation to a noticeable degree. To overcome it, we inserted a tiny amount of assembly code in a secondary implementation. Our assembly code focused only making sure that 32 X 32 bit multiply did not magically turn into a 64 X 64 bit multiply.

	32-bit Machine		64-bit Machine
	Optimized	Assembly	Optimized
SANDstorm -224, -256	4600	4000	2340
SHA-1	1200		930
SHA-256	2600		2500
SANDstorm-384, -512	38000		12200

Figure 7: Timings in clock cycles of a single compression operation.

Again, since assembly versions of the algorithm were not to be a priority in the Round 1 of the competition, we did not include an assembly version for SANDstorm-512, nor did we port the small amount of assembly code to the 64 bit machine. In any event, we feel further optimizations are available with or without assembly.

Operation Counts
SANDstorm-256
During the operation of SANDstorm, there are several different logical operations that we will group into a single cost category. These are not always entirely the same, but close enough for this discussion. Logical operations include: Not, XOR, AND, Shift Left, Shift Right.

There are two arithmetic operations: addition and multiplication. The additions are either modulo 2^{64} or 2^{32}. On some architectures there may be a significant difference so, since the majority of the additions are 64 bits, these will be specified as adds, or +. The 32 bit additions will be called, 32 bit adds. The multiplications * are 32 x 32 to 64 bit computations.

Similarly the BitMix function as written takes 28 logical operations, but it can be completed in 16 by doing the following four steps in turn
$T = (A \oplus C)\&6666666666666666$; $A = A \oplus T$; $C = C \oplus T$;
$T = (B \oplus D)\&3333333333333333$; $B = B \oplus T$; $D = D \oplus T$;
$T = (A \oplus B)\&5555555555555555$; $A = A \oplus T$; $B = B \oplus T$;
$T = (C \oplus D)\&5555555555555555$; $C = C \oplus T$; $D = D \oplus T$;
where the numeric constants are represented in hexadecimal notation.

- $ROTL^n(Z)$ requires a left and right shift and an XOR for three logical operations
- $F(Z) = X^2+Y^2$ modulo 2^{64} requires one + and two *.
- $G(Z) = X^2+Y^2+ROTL^{32}((X+a)(Y+b))$ modulo 2^{64}
    - X+a and Y+b are 32 bit additions. Additionally there is one rotate (3 Logical), three multiplications, and two 64 bit additions.
- $Ch(A,B,C) = (A\&B)\oplus(\neg A\&C)$ takes four Logical operations as written, but it can be written as $Ch(A,B,C) = C \oplus (A\&(B \oplus C))$ which is 3 logical operations

- $SB(Z) = Z$ is an AES sbox look-up and replace. Our implementation requires two Logical operations and one look-up.

The round function operates on four words in turn and then performs the BitMix operation. Recall each step in the round function (with the non-essentials for counting stripped out) is:

$$ROTL\ (SB\ (W_i + F(W_{i-1}) + Ch(W_{i-1},W_{i-2},W_{i-3}) + A(r,i)))$$

The message schedule repeats the following computation 25 times:

$$ROTL\ (SB(d_{i-8} + G(d_{i-1})+Ch(d_{i-1},d_{i-2},d_{i-3}) + d_{i-4} +B_i\ ))$$

The message schedule and the chaining variables are XORed into the state variable, additionally there are a few extra operations required for the Round 0 input.

The following table lists the operations for a single compression step.

	Logical		64 bit +		*		Look-up		32 bit +	
Round Function										
ROTL	12									
SB	8						4			
Additions			12							
F(Z)			4		8					
Choose	12									
BitMIX	16									
Total in one Round	48		16		8		4			
Total in five Rounds		240		80		40		20		
Message schedule										
ROTL	3									
SB	2						1			
Additions			4							
G(Z)	3		2		3				2	
Choose	3									
Total in one Step	11		6		3		1		2	
Total in 25 Steps		275		150		75		25		50
State variables										
Round 0	36									
Rounds 1-4	32									
Total		68								
Total for one compression		583		230		115		45		50

Figure 8: Operation Counts for a Single Compression Step

On a 32 bit machine, if the logical operators and the 64 bit additions take twice as long as a 32 bit addition and if each multiplication takes 3 times as long as the 32 bit additions, and a table look up counts the same, then we have 2066 32-bit instructions.

On a 64 bit machine, if the 32 bit additions and look up take as long as a 64 bit addition and the multiplication takes 3 times as long, then we have 1284 instructions.

The operations in Figure are for a single compression and do not account for the mode. The SANDstorm mode includes up to five levels including a finishing step, which always occurs. Each time a superblock is begun, an additional 20 or so XORs for initialization must occur. Depending on the size of the superblock these may be in the computational noise.

The effect the mode has on the total operation counts depends on the length of the messages. For very long messages the finishing step may be amortized and the overall effect of the mode is an 11% increase in counts.

For one block messages the finishing step will double the number of operations. For two block messages the overhead is 50% reducing rapidly (but not consistently) with message length down to the 11%.

Cost Estimate of the SANDstorm Hash for an 8-bit Processor

We selected the Z80 architecture for our estimate. The Z80 is a well-known architecture (see http://en.wikipedia.org/wiki/zilog_Z80), is about 30 years old, and is available in a variety of implementations and simulations, including as an FPGA. The speed, the number of clock cycles, and the relative timing for the instructions are all platform dependent. Our performance cost estimate is simply a count of the number of instructions executed in a reasonable implementation of SANDstorm on the Z80. The simplicity of the instruction set makes the instruction count a fairly platform-independent performance measure.

Our estimate of the number of instructions required to compute the SANDstorm hash on a Z80 microprocessor is based on the following obervations: For a 224 or 256 bit output, each use of the compression algorithm takes about 50,000 instructions when processing a block of 512 bits. For a minimal message of up to 511 bits, two calls to compress are made, so a minimum hash will take 100,000 instructions for a 224 or 256 bit output.

For a 384 or 512 bit output, each use of the compression algorithm takes about 160,000 instructions, processing a block of 1024 bits. A minimal message of up to 1023 bits will take 320,000 instructions for a 384 or 512 bit output.

For a long message, there are 1.1 compress calls per block. The cost of a 224 or 256 bit output hash is about 110 instructions per input bit. The cost of a 384 or 512 bit output hash is about 350 instructions per input bit.

The details of our estimate are:

Moving a 64 bit quantity: 11 instructions. (3 setup, 8 data moves)
Adding two 64 bit quantities: 26 instructions. (2 setup, 8 sequences of load, add/adc, store)
XOR of two 64 bit quantities: same as Add.
Rotating a 64 bit quantity, one bit position: 10 instructions.
Multiplying two 32 bit quantities, producing a 64 bit product: 270 instructions (average).(Note, this is based on the comb algorithm: The multiplicand is added into the product register with any of four byte offsets, controlled by bits in four bytes of the multiplier. The product register is then shifted left one bit, and the conditional additions are again performed, controlled by another four bits of the multiplier. Eight cycles of this process develops the complete product.)

In SANDstorm-224/256:

The F function (within the round function): 600 instructions. (Two multiplications, one 64 bit addition.)

The G function (within the message schedule): 950 instructions. (Three multiplications, two 32 bit additions, two 64 bit additions.)

The Ch function: 40 instructions. (CH(A,B,C) = C⊕ (A&(B⊕C). Load,Xor,And,Xor,Store, 8 times.)

One 64 bit word of the Message Schedule: 1200 instructions.

Bitmix on four 64 bit words: 500 instructions.

The message schedule: 30,500 instructions. (Twenty five result words, one Bitmix.)

One 64 bit word within a round of the compression function: 750 instructions. (Three 64 bit additions, one call each to F and Ch, one sbox lookup, one 1-place rotation of a 64 bit quantity.)

One round of the compression function: 3800 instructions. (Four mixing operations, one Bitmix, two or three xors of 256 bit words.)

Compressing one 512 bit block: 50,000 instructions. (Message schedule, five rounds of compression.)

For Sandstorm-384/512:

The computation pattern of SANDstorm-384/512 is the same as SANDstorm-224/256, but the operands are twice as long: 128 bit arithmetic replaces 64 bit arithmetic. For most operations, this simply doubles the number of Z80 instructions required. However, the multiplication operation is different: The cost of 64x64 bit multiplication is about 3.5 times the cost of 32x32 bit multiplication when using the Karatsuba algorithm.
In SANDstorm-224/256, 75% of the work is in the multiplications. To estimate the cost for SANDstorm-384/512, we split the work of SANDstorm-224/256 into multiplication and non-multiplication parts, (37,500 + 12,500), and scaled by 3.5 or 2 respectively. The total is about 160000 instructions, to run the compression algorithm for a 1024 bit input block.

For 8 bit processors, most of the work goes into multiplications. A processor with a hardware multiplication, such as the old M6809, will be much faster. For the Z80, good results might be obtained with algorithms such as Quarter-Squares or Difference of Triangles, which use modest size tables to speed up multiplication.

11. Memory Usage

There are several ways to implement the SANDstorm family; some require more memory than another. This discussion focuses on a reasonable software implementation. SANDstorm-256 uses 50 of the 64 32-bit constants used by the SHA family, during the compression operation. SANDstorm also uses the same eight initialization constants as the SHA, these constants are expanded into five 256 bit initialization constants per level. Eight additional fixed constants are used. Two of the additional constants are 32 bits each and 6 are 64 bits. A reasonable software implementation of SANDstorm would precompute and store these constants. This is a total of 50*32 +5*5*256+448 = 8448 bits. A more conservative approach needs only the additional 448 constant bits that are separate from what is needed for an implementation of SHA-256. Of those 448 bits, 4*64 bits are the selector bits in the BitMix function. These have a very simple bit pattern that may be recreated when needed to reduce the fixed storage.

Both the round function and the message schedule use the AES sbox. There are 256 one byte entries. From a storage standpoint, an implementation of the SANDstorm algorithm has a high probability of being combined with AES encryption, so the sbox should be available for use, thus, possibly, reducing the total memory usage. Total 2048 bits.

The message schedule computes 25 64 bit values after the 512 bit message is input. These 25 values can be unrolled and stored or computed as needed. If completely unrolled and combined with the input message, there are 33*64 = 2112 bits. On the other hand, the message schedule may be thought of as a block of eight 64 bit words and processed in an as-needed fashion. In this case there are only 512 bits to store.

In the compression function, there are five rounds, each with a chaining variable that is 256 bits in length. (One of the chaining variables is actually a constant for a given superblock). Each of the five levels in the tree requires five chaining values that must be manipulated during the course of the algorithm. That is 5*5*256 = 6400 bits. However, Level 0 must be completed before Levels 1, 2, and 3 can begin. The values from Level 0 are used as part of the initialization of the chaining values for those levels. Similarly, Level 4 is not invoked until all other levels are complete. At any given time at most three of levels require storage of the chaining values. That is 5*3*256 = 3840 bits.

Data is also passed to from one level of the tree to the next. This requires at most 2 512 bit values in addition to the message blocks being processed in Levels 0 and 1. The data for Level 4 does not get created until Level 3 is completed. So, a total of 1024 bits must be passed from level to level.

The round function actively operates on four 64 bit state words at time, thus requiring 256 bits.

	Constant	Volatile	Active	
Constants	8448			
AES sbox	2048			
Message Schedule		512		
Chaining Variables		3840-6400		
Level Data		1024		
State Words			256	
Totals	10496	5376-7936	256	16128-18688

Figure 9: Memory Usage in bits

Depending on the implementation, the total amount of RAM for function variables is about 2 KB.

Note that short messages need less memory. For one block messages, when Level 0 completes, the output can go directly to Level 4. At any given time only one set of five chaining values needs to be retained: 4352+512+5*256+256 = 4352+2048 = 6400 bits.

Similarly, shorter messages will not use Level 2 or 3 and so will use fewer resources than longer messages. One would expect memory requirements to be around 4352 bits for fixed constants and between 2048 and 8192 additional bits required for processing, depending on message size and implementation.

SANDstorm-224 storage requirements are the same as SHA-256. The constants are the same except for an additional eight 32 bit initialization values. SANDstorm-512 and -384 require approximately twice the storage as SHA-256 and -224.

Our reference implementation including `.c` and `.h` files including the handling api is on the order of 60 KB. A Linux executable is on the same order of magnitude in size. No effort was spent trying to minimize these numbers.

Appendix A Sample Test Vectors

All values in this appendix are represented in hexadecimal notation. A full listing of all required test vectors is included on the optical media as required.

message = abc
224: fd76ce6091725130 cfe7248a1b1db4eb b498dbb351dfcce6 e681e46a
256: 10c9c33e26f42840 305d5d0a7b437809 777e904d8f9f1a3a 2dd0de51c555f2ef
384: 4f0fddb20f630bad c4929a5744fa645d 01a73a42dbacbd9e 5d19668eeb218270 2d95c810f7d0ca23 c9a620c8ecf58e31
512: b8166d6e33c8954f 9c3daf42b3e35e72 051d577eed8287e3 01e0acdb20cfdffb 8777aec90553cc28 d31be552f941ff80 097beac52d8adc2f 0139ba69e2111008

message = abcdefghijklmnopqrstuvwxyz
224: 307bb0ef1399ec82 7e4dc8099833f1c5 1d2110b9fbf44a73 efd85656
256: 51e5ff14342d4440 2224d832d2d674e8 3241c98ade5408dd 2dfd5e069d4a4b70
384: 18c96b6c274e67c2 dc7a0ffd47f3c242 bddf7a5dd3197cca cf521635f56ae8d5 e3ff63df85eb7bae 6d2fbee6162abcc2
512: 67da9b09e9fbe195 b738897153c9e0ac d85916084c9b9517 28cd08bd53aefb2c 557f7a8088972673 b75a9b069fe2d2e5 669c3c7bf7d16f50 3a8ec4fd6167e99b

message = 100 alphabets
224: 522be4eef140135e 2452e210149dbcbe d71598627565b462 90373784
256: 467390f36e287494 f9c732f9ae9e3499 af83e2d7064a8f2d a9acdf50d3865cf9
384: 9f6cc349a69c930e 7ef407ecde5e0e51 396b5fa682c4511a c7fcbfda166ded48 896d444b4e49e22c 034e7bc2ec3fb8b7
512: 71b416b9ff1ae00a 24c4b2eb5b0cc443 30d5704af2ffcc2d 670daf227ef5c8c2 b4a2911594306c32 50d0add93e3b4c82 2fb1b12b09f6741a c020f86051306f48

message = a, one million times
224: 2adbef88964d53aa 5c050a3c6c1028d1 26a4fd6f64f8335f 8684fe50
256: bb653933aad7cc82 cef83991b4e2db24 5ef608d440eeaf09 90d69d8e27c265da

384: 946514f9d42b3826 cd549b26c2eecc73 c9dc8fd9a1e857d2 4826ee2a14d008a7 ec6fe379f4a931b9 199e7655ec8adadf
512: a3aad31a418ebd58 a93a9a055ecce4d1 81c63f9f4a628b83 87b529a5987ad88d dccd301286ca647d deb09f80e920f1c0 db3665a4493ef56d 8605a2c9a8c88b09

message is null
224: f351cc5f721dbf13 ca9086630c07112e 71f96c7e13a0bea3 879ccacd
256: 7325f39f1c05fe93 4064afd4513e0ce6 49ffb671f0c80983 6f65921dd36b2399
384: 2a8469e051340868 aca6ff9a9ce7ec9e e9074cfb1ab7c0d7 8b87ca51589897b0 6f33cf8bd40b4204 4b2fecb3ee8bcba2
512: 7bc6848a21a1fbd6 8eeb18a7fcca5734 ba005835406a5b5c ddb199f94b26044f da7b6121410322f5 b0efcdc31df9a78d 61d25ab949c7066c d6664f4f6b200ce7

message is a single 1 bit
224: 3f0d6973ba848986 62f52ccddd551f02 b36611832114bcf4 c17d0cc0
256: 300ad96fb1a2934f c78497abae9880ed fa76ebf870cc3a9a d75a803bf9b953b7
384: eb02a645ed3e7bfd ccfb59a920bb5fba c442797b260ac66f 0618e3a54d0d2e42 a57833d206648af3 293602cb6b582f6d
512: 5ce44321a52650d5 4f69a4b8521e3a57 4715e768f5f68cb2 1cc9b95668d4ea35 53e35734b50957f3 8d8fb433b4deb12c 2510bb5904748fa4 f3194dbbe5d6b30e

TunableSecurityParameter = 2

message = abc
224: eb7f445967d67c50 4b8d4b2a21ddc126 3ae72ec74f202492 ffa745ac
256: 5128ea92679baa58 9a8299ff5df27584 825f593c1096b917 e7d399dfbfc484f1
384: 037b25d2574dbd50 89a3cfc667609795 8e9cdc64661639ca b75c29581bfc15db 9dd83d4a1ebd51b8 6c4b149472d66142
512: 3d29a8bbb3fedbc1 639b888c10efca94 105618c9ace6613b 2097945f2c4536a7 ef61c50dc7983ca4 1ba54d62c695df8d 5786a095f664b30c fa94e5f743fbbe33