SAND2008-7086P

The SANDstorm Hash

Authors
Tim Draelos, Richard Schroeppel and Mark Torgerson

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

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.

1. Introduction

Our overall design goal was to make a very strong hashing function that performed reasonably well on the most common architectures. We also had other desires for the SANDstorm family, such as:

- Parallelizablility and/or pipelineablility
- Deviation in structure from the MD/SHA family designs
- A finishing step
- Internal state larger than the final output of the hash to add resistance to attacks on very long messages as well as attacks focusing on multicollisions and herding.
- Reusage of the SHA family constants so that implementations that must support both the SANDstorm family and the SHA family would be able to reuse those constants as needed.
- Use the strategy for obtaining 224 bit hashes as is used in the SHA family. 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.
- Be compatible with the NIST methods of HMAC, randomized hashing schemes. This will allow a "plug and play" with many of the data formatting mechanism and program wrappers currently in use.

Our design goals have been realized. Our algorithm has a great deal of mixing and its performance is on par with the SHA family of algorithms on the 32 bit architectures that we tested. Significant speed gains can be realized on 64 bit architectures. If a software implementation includes a tiny amount of assembly code considerable speed enhancements can also be found.

The real performance benefit of our algorithm is that it is designed so that there are two different types of parallelization available. The compression function is parallelizable to a large degree and the mode in which it is used also is designed especially to allow a large degree of parallelization.

Design Overview

The SANDstorm-256 and -224 Hash processes 512 bit blocks and operates on 64 bit words. Whereas the SANDstorm-512 and -384 Hash processes 1024 bit blocks and operates on 128 bit words. Each uses a mode that is a modified and truncated k-ary tree with a finalization step. Internal to the mode is the SANDstorm method of chaining, which does fall under the Merkle-Damgard umbrella, but is different from the standard implementation. We also have an efficient compression function.

The SANDstorm hash is compatible at the function level with the SHA family. In particular it may be used with the typical HMAC methods such as those in FIPS 198-1, The Keyed-Hash Message Authentication Code (HMAC), July, 2008 found at http://csrc.nist.gov/publications/PubsFIPS.html, and randomized hashing methods

such as those described in SP 800-106, DRAFT Randomized Hashing Digital Signatures (2nd draft), July, 2008 and found at http://csrc.nist.gov/publications/PubsDrafts.html

There are four main components in the design

- Padding
- Mode
- Chaining
- Compression

Each of these will be explained in turn.

Notation and Conventions

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

We felt that for the sake of consistency we should use the same notational conventions. There are a few rearrangements and deletions of the FIPS text along with a few additions as needed for the SANDstorm hash.

Symbols

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

^ Bitwise AND operation

⊕ Bitwise XOR ("exclusive-OR") operation

¬ Bitwise complement operation

+ Addition modulo 2^w

* Multiplication modulo 2^w

<< 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.

The rotate left (circular left shift) operation, $ROTL^n(x)$, 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.

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

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. So 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 such as $E=(e_0, e_1, e_2, e_3)$.
b) For SANDstorm-384 and SANDstorm-512each 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 padding for SANDstorm is simple and unambiguous. We pad each message by appending a 1 to each message and then filling with 0’s until the result is a multiple of the block length.

Let η be the length in bits (before padding) of the message to be hashed. If η is a multiple of the block length, then a 1 followed by the block length minus one number of zeroes

will be added, which increases by one the number of blocks the message and padding is be parsed into. Otherwise, the padding will not result in an extra block.

The message length in bits η must be passed into the mode which uses the length in the finishing step.

The SANDstorm hash allows for $0 \leq \eta < 2^{128}$ no matter the hash output size.

3. The SANDstorm Mode

One downside to a typical Merkle-Damgard construction is that parallelization and/or pipelining is not readily possible, thus limiting the throughput of high-end implementations. On the other hand, tree based hashing is highly parallelizable, but the depth of the tree and, the amount of storage needed are variable depending on the size of the message. 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 based approach that is partially parallelizable but has a bounded depth and storage. The amount of storage is at most 8 blocks of intermediate data, with careful message processing. Yet, given the right resources, there is a tremendous opportunity for parallelization of on the order of a factor of 1000.

Each level of the tree uses the SANDstorm chaining described below to build the k-ary tree. The mode also provides for a finishing step as well as an early out option to aid in the performance of short messages.

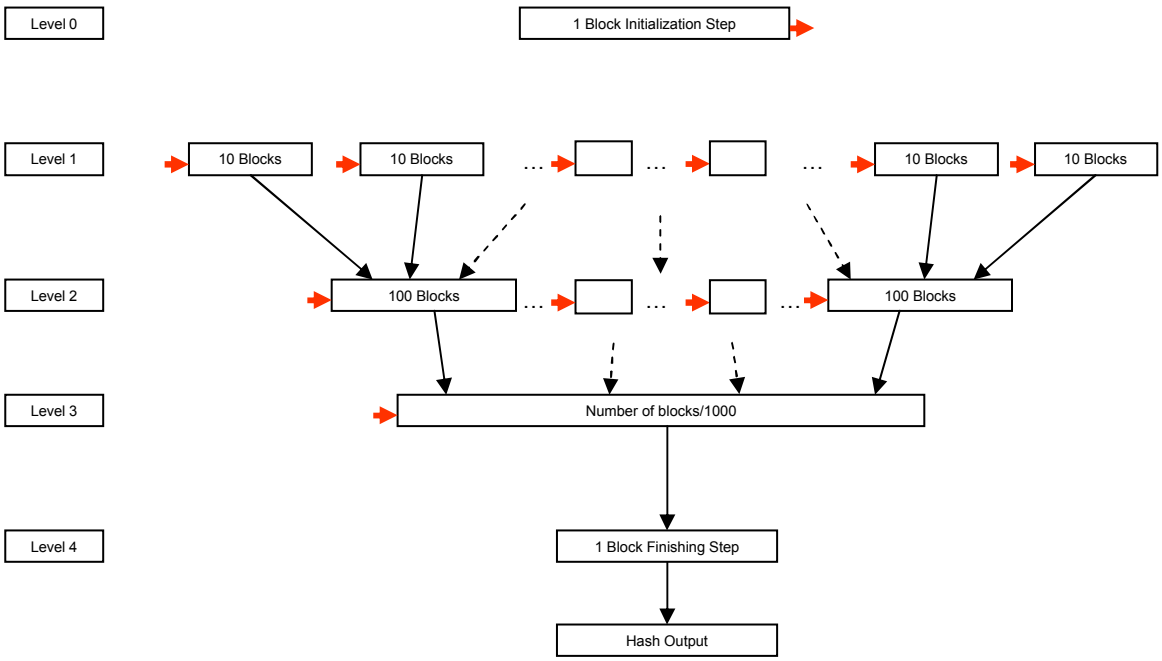

Figure 1: SANDstorm Mode

SANDstorm Mode Description

For this discussion it is assumed that the input message will already be padded and parsed into an integral number of 512(1024)-bit blocks.

The SANDstorm mode, nominally has five different levels.
Level 0 and Level 4 are completed no matter the length of the input message and each process a single block.
Level 0 is somewhat like a conditioning step and its output is used in all other levels.
Level 4 is a finishing step.
Levels 1, 2, and 3 are used depending on the length of the message.

Each level of the tree combines data into superblocks (a grouping and chaining of a certain number of data blocks) the size of which depends on the level. The initialization of each superblock will include a superblock number. The final output of each superblock is the result of the chaining operations and is four 256(512) bit strings. Those strings are combined to produce a new data block of 512(1024) bits. The new block is either fed into the next level of the tree or fed directly to the finishing step depending on the message length.

Let $M=M_0,M_1,\ldots,M_m$ be the input message parsed into m+1 512(1024) bit blocks, after padding.

If m=0, then Level 0 processes M_0 and passes the output to the finishing step, Level 4. This passing of shorter message data to Level 4, rather than through the rest of the tree, we call an "early out".
If m>0, then Level 0 and Level 1 combine to partially compress M to create a new message $N=N_1, N_2,\ldots,N_n$, where the N_i are 512(1024) bits in length. In particular, the M_i are parsed into "superblocks" of size 10 and processed sequentially with the SANDstorm chaining to create one block of N. That is, for i=1 to n the message blocks $M_{(i-1)*10+1}$, $M_{(i-1)*10+2}$,…, and M_{i*10} are processed with the SANDstorm chaining method. The final chaining value of the superblock is the four 256(512) bit strings:

$$\text{chn}(0,i*10),\ \text{chn}(1,i*10),\ \text{chn}(2,i*10),\ \text{chn}(3,i*10),$$

where each chn(j,i*10) is a 256(512) bit string. How these are computed will be explained below and from which we compute:

$$N_i=(\ \text{chn}(0,i*10) \oplus \text{chn}(2,i*10),\ \text{chn}(1,i*10) \oplus \text{chn}(3,i*10))$$

N_i is a function of M_0 via details from the compression function. If m is not an integer multiple of 10, then N_n is produced from SANDstorm chaining the last (m modulo 10) blocks of M and XORing the four output values as is shown above.

If n=1, then N_1 is fed directly into the finishing step of Level 4. Otherwise, the message N is partially compressed in Level 2 to produce $P=P_1, P_2,\ldots,P_p$, where again the P_i are 512 bits in length. Here we have that 100 blocks of N are used to produce one block of P. In

particular, for i=1 to p the data blocks $N_{(i-1)*100+1}$, $N_{(i-1)*100+2}$,…, N_{i*100} are chained together with the SANDstorm chaining to produce the P_i, in the same fashion that the N_i are produced. The four word output of the superblock are paired together and XORed as above. 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.

If p=1, then P_1 is fed into the finishing step of Level 4. Otherwise, the message P is fed into Level 3, which SANDstorm chains completely. The length of P is bounded by $(2^{128})/1000$. The final four 256(512) bit strings are paired and XORed as above and fed into the Level 4 finishing step.

For SANDstorm hash-224, the final message digest is the XOR of all four 256 bit strings output 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 strings output of the Level 4.

For SANDstorm hash-384, the final message digest is the XOR of all four 512 bit strings output 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 strings output of the Level 4.

The messages N and P are, in a sense, artificial and need not be fully created before the algorithm begins processing the next level. As soon as the appropriate data is available, a level may begin its processing. Morever, if an application can guarantee that its messages will always be less than 12 blocks in length, its implementation of the SANDstorm need not include code for Level 2 or Level 3. Similarly, Level 3 can be eliminated if all messages are less than 1002 blocks in length.

Levels 2, 3, and 4 will always receive the 512(1024) bit message blocks. No padding is necessary on those levels.

SANDstorm Mode Performance

The SANDstorm mode is designed to be partially parallelizable in the first two levels of the tree. Levels 1 and 2 have Merkle-Damgard chaining within the superblocks of size 10 and 100, respectively, but the superblocks themselves can be processed independently. Level 3 is processed sequentially in the Merkle-Damgard fashion. For large messages, this allows a maximum speed up of 1/(1000) due to parallelization in the mode.

There is a good deal of flexibility in the construction and the available factor of parallelizability depends on the size of the message being hashed. If one has a message 12+ blocks in length, there is an opportunity to process two Level 1 superblocks of size 10, independently. Such would give a speed up of roughly a factor. Similarly, if one has a message of greater than 1002 blocks in length, then one can begin taking advantage of the

parallelization in Level 2, with benefit ranging up to a maximum of a factor of 1000 depending on resources.

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 mechanisms for short messages, the effect of the levels of the tree, including the impact of the finishing step, is mitigated to some extent. If one has a message of length $0 \leq \eta < 512(1024)$ in bits, then the message will pad out to exactly one block. So then Level 0 will operate on the message and then the output will pass directly to Level 4 and so two compressions are needed to process a message of that size. A message that is between $1 \leq k \leq 11$ blocks long would require k+1 compressions including the finishing step.
The second level of the tree is not invoked until the message becomes at least 12 blocks in length and would require 14 compressions. At that point, the message would not be considered extremely short and the impact of the additional level is mitigated to some degree by the length. Similarly, Level 3 is not invoked until the message is at least 1002 message blocks long.

There is a certain amount of 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 has some other flexibilities associated with precomputation. 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 to some degree, say always contained in the input of a superblock 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 process may process up to the changed position, reducing the amount of needed storage and the computations needed.

Choice of Superblock Size

The choice of a superblock size of 10 and 100 is somewhat of an arbitrary choice. If one were to reduce the size of the either superblocks or both, then the maximum amount of

resources needed to take full advantage of the mode would be less, but the maximum potential advantage would be less as well. On the other hand, we could have added more levels in the tree or increased the size of the blocks to increase the potential for parallelizability. However, the resources needed to take full advantage could quickly become absurd. Our choice is based on what we considered an upper bound on what might be reasonable for a very high end implementation. In this way, one may pick the amount of resources they are willing to apply and, for any reasonable choice, one may see performance benefit associated with that choice.

If the SANDstorm hash survives the competition and is chosen by NIST, further discussion of the choice of superblock size might be in order. Even though we would support a discussion of changing the superblock size to some other fixed value, it is another issue to have a variable superblock size. It does not seem to matter what the superblock sizes are from a security standpoint, but when one is able to freely adjust both message and superblock size one must analyze all possible combinations to determine if something untoward is going to happen. We know of no such vulnerabilities, but as a rule having too much variability in an algorithm may induce vulnerabilities associated with those variabilities.

4. The SANDstorm Chaining

In the usual iterative description of the Merkle-Damgard construction, one computes a chaining variable as $h_i=H(h_{i-1},M_i)$. So that the value of h_i is a function of h_{i-1} and the message block M_i. Almost all implementations XOR inputs and outputs such as setting $h_i=H(h_{i-1} \oplus M_i)$, which is simply not compatible with parallelization and/or pipelining. So, to achieve some level of parallelizability within the chaining, we deviated from this usual implementation and leveraged the more general form of the Merkle-Damgard construction.

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

Our nonstandard view of $h_i=H(h_{i-1},M_i)$ allows us to parallelize within the chaining of the superblocks and allows us to carry more internal state forward and yet remain efficient.

SANDstorm Chaining 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 down at the superblock level and show how to pass data from one level of the tree to the next.

Level 0
Level 0 operates on a single block and is somewhat different than the other levels.

The compression function nominally has 5 rounds. Each of those rounds will see an input from a bank of constants and from the message schedule. The input constants for each hash size C_0, C_1, C_2, C_3, and C_4 are defined below.

The output of Level 0 is four 256-bit strings S_1, S_2, S_3, and S_4. The four values will be part of the 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 sent directly to Level 4 to comprise the input message (s_5,s_6) for that level. Where

$$s5=S_1 \oplus S_3 \text{ and } s6=S_2 \oplus S_4$$

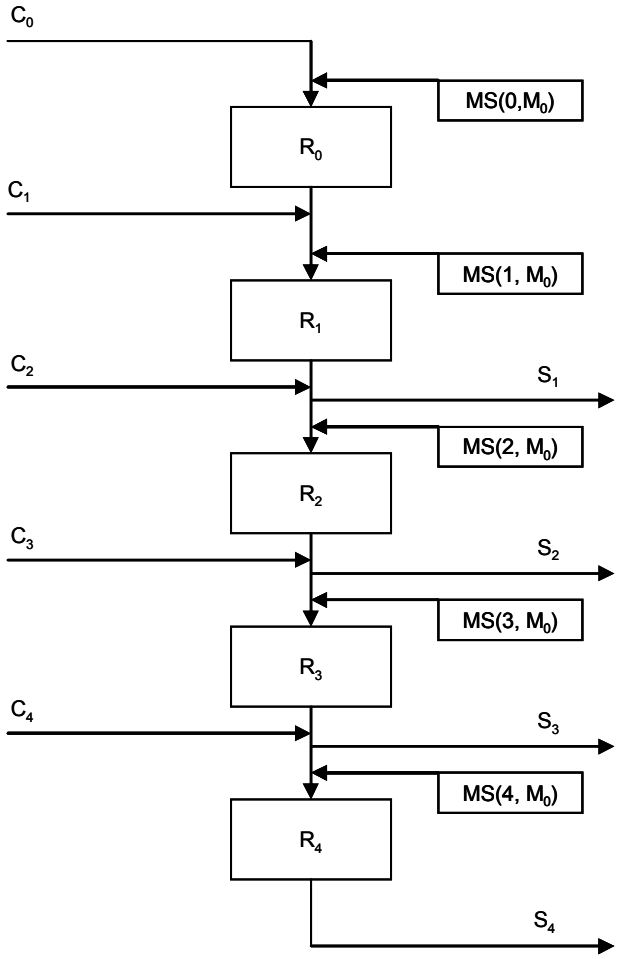

Figure 2: Level 0

- R is the round function.
- MS(r, M_0) is the message schedule with a 256(512) bit output, which is a function of round r and data block M_0.

- Both 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 inputs 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 since the other levels may not actually operate on the message blocks directly.

Levels 1 ,2, and 3
Levels 1, 2, and 3 chain multiple blocks of data together. 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 constants for the next block. So, 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 shows the state transition from one block position to the next given a superblock of size two.

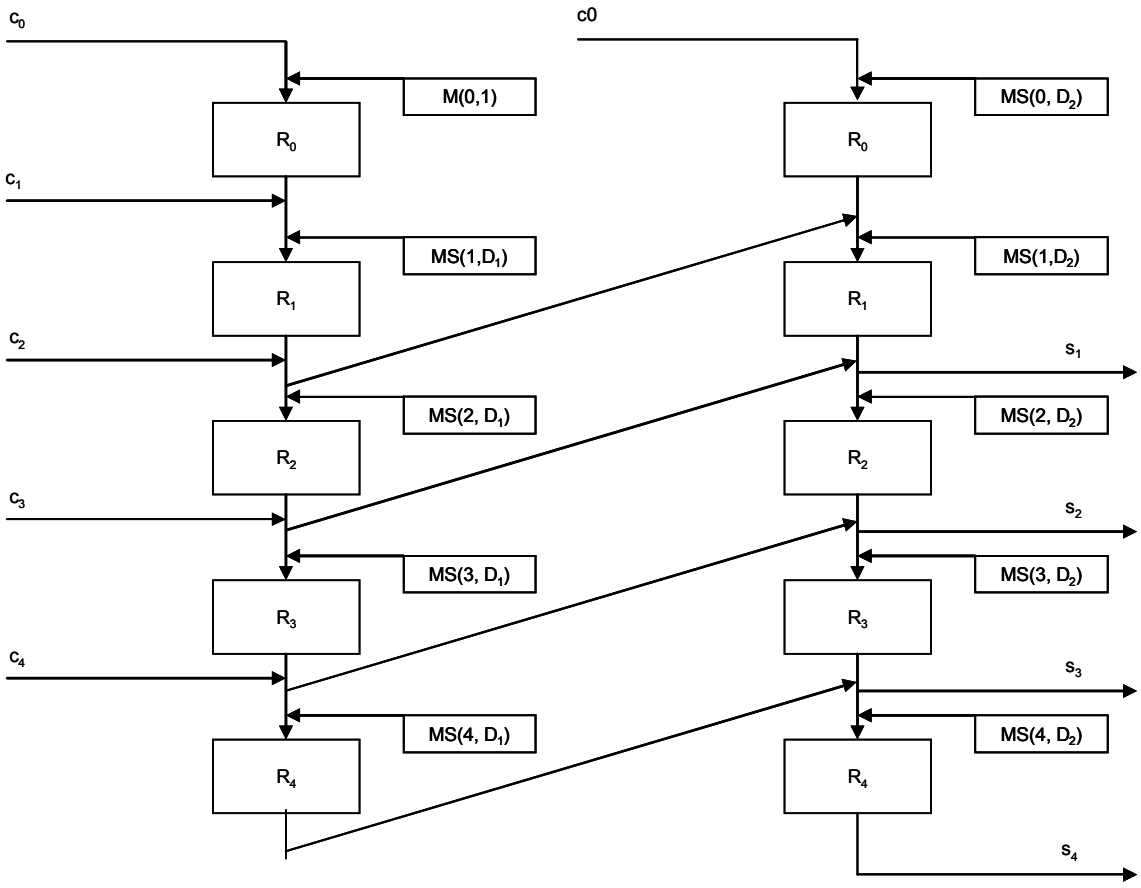

Figure 3: Two Block Superblock

Figure 2 represents a superblock of size two. To extend to superblock of larger size one continues the construction by taking the output state of one round and applies it to the input state for the same round for the next block.

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.

As with Level 0, to initialize the compression of the superblock there are input constants c_0, c_1, c_2, c_3, and c_4. These constants are a level dependent function of the Level 0 input constants and the Level 0 output state. For a given level in the SANDstorm mode tree the value c_0 is fixed. For Levels 1 and 2 the superblock number is also included in the computation of the initialization constants. The specific constant values are given below.

For a given level and superblock position, let chn(i,k) be the chaining value from round i at block position k. The input constant for the superblock is then c_i=chn(i,0). Similarly, let R(i,k) be the output of round i at block position k.

For $1 \leq i \leq 4$ and $1 \leq k \leq j$ the input into round i at block position k is
$R(i-1,k) \oplus chn(i,k-1) \oplus M(i,D_k)$

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

The end of the superblock compression produces four 256(512)-bit values s_1, s_2, s_3, and s_4 that are combined together to produce the data (s5,s6) for the next level in the tree. The pair of values

$$s_5 = s_1 \oplus s_3 \text{ and } s_6 = s_2 \oplus s_4$$

are 256(512) bits in length values which are treated as a 512(1024) bit data block (s5, s6), which is fed into the next level of the tree.

Level 4

The finishing step of Level 4 is similar to Level 0 in that it operates on a single block of data.

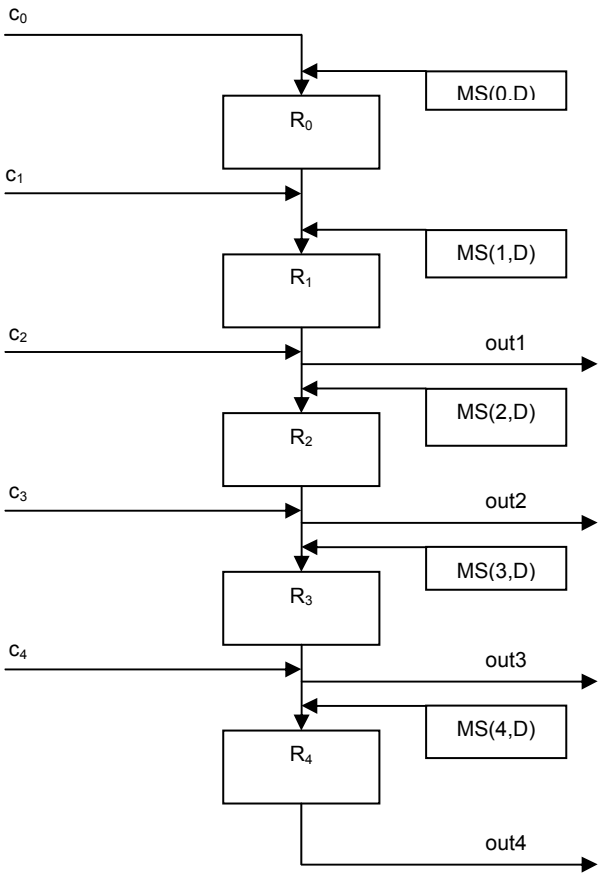

Figure 4: Level 4

The input constants are defined below, but are a function 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 output combines together to create the final hash value of the message M.

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

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.

Again given message $M=M_0,M_1,\ldots,M_m$.
For SANDstorm-256 and -224, Level 1 processes 10 blocks at a time. Then with $M_{(i-1)*10+1}$, $M_{(i-1)*10+2}$,…, and M_{i*10} the counter i is 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 $1 \leq i \leq k$. SANDstorm-512 and -384 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 derivates 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 $1 \leq i \leq k$. 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. Given in hex:
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. Given in hex:
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 Constants

C_0, C_1, …, C_4 each are 256(512) bit words comprised of the SHA initialization constants. Where H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7 are those constants.

$C_0 = (H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7)$
$C_1 = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_0)$
$C_2 = (H_2, H_3, H_4, H_5, H_6, H_7, H_0, H_1)$
$C_3 = (H_3, H_4, H_5, H_6, H_7, H_0, H_1, H_2)$
$C_4 = (H_4, H_5, H_6, H_7, H_0, H_1, H_2, H_3)$

The following are the initial values for SHA-224, which are given in hex. These are used to create the C_i for SANDstorm-224.

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 hex, and which are obtained by taking the first thirty two 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 hex. 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 hex. 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 more standard implementations of the Merkle-Damgard construction, one must wait until one block of data is completely finished before beginning the processing of the next block of data. 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. However, if one has sufficient resources available, and can take advantage of the connections between rounds there is the possibility of having a partial pipelining. That partial pipelining will have latency of a single round rather than the latency of all 5 rounds needed for a typical Merkle-Damgard construction.

For the SANDstorm compression function, about 2/3 of the work is done in the message schedule which can be parallelized and pipelined, so if the resources are available, one should be able to see a factor of 3*5=15 increase in speed over sequential processing of

the data blocks in the standard Merkle-Damgard fashion. This does not include any gains one might see by including additional resources to speed up the round function itself.

5. The SANDstorm Compression

The SANDstorm compression function has two main ingredients, the round function and the message schedule. The SANDstorm round function is a one-to-one function of 256(512) bits operating on four 64(128) bit words. The round function is divided into 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

General Functions

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

- $ROTL^n(Z)$=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 Y+a and Z+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)⊕(¬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.

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

BitMix Function

The bit mix 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 hex, these are:

J_8 = 0x8888888888888888
J_4 = 0x4444444444444444
J_2 = 0x2222222222222222
J_1 = 0x1111111111111111

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) ⊕ (J_4 & B) ⊕ (J_2 & C) ⊕ (J_1 & D)
B' = (J_8 & B) ⊕ (J_4 & C) ⊕ (J_2 & D) ⊕ (J_1 & A)
C' = (J_8 & C) ⊕ (J_4 & D) ⊕ (J_2 & A) ⊕ (J_1 & B)
D' = (J_8 & D) ⊕ (J_4 & A) ⊕ (J_2 & B) ⊕ (J_1 & C)

Round Function

The round function 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.

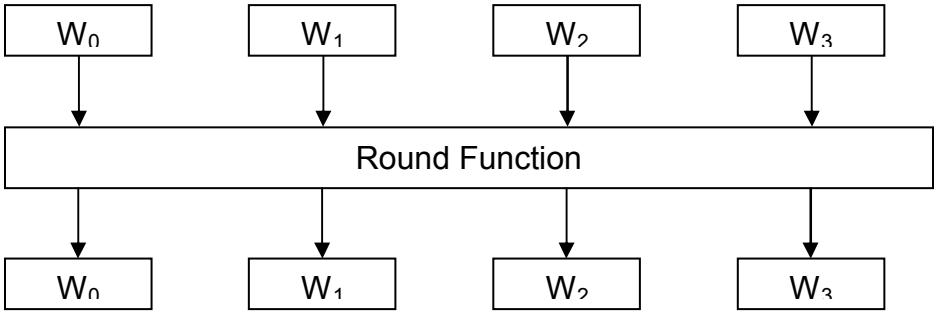

Figure 6: Round Function

For i from 0 to 3

Set $W_i = ROTL^{25}(SB([W_i + F(W_{i-1}) + Ch(W_{i-1},W_{i-2},W_{i-3}) + A(r,i)] \text{ modulo } 2^{64}))$

Set $(W_0, W_1, W_2, W_3) = BitMix(W_0, W_1, W_2, W_3)$

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 four and the computations of W_i are assumed to be iterative, so when each value is updated the updated value is used to update subsequent values. As mentioned the BitMix function operates on the words in parallel.

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 34 64 bit words.

For i from 8 to 32

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

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.

$MS(0,D)=$

$BitMix(ROTL^{19}(d_0) \oplus d_4, ROTL^{19}(d_1) \oplus d_5, ROTL^{19}(d_2) \oplus d_6, ROTL^{19}(d_3) \oplus d_7)$

$MS(1,D)=(d_{14}, d_{15}, d_{16}, d_{17})$

$MS(2,D)=(d_{19}, d_{20}, d_{21}, d_{22})$

$MS(3,D)=(d_{24}, d_{25}, d_{27}, d_{27})$
$MS(4,D)=(d_{29}, d_{30}, d_{31}, d_{32})$

Constants

SHA-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, …, K_{48}. In hex, these 48 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
```

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 $B_i=K_{2j}*2^{64} + K_{2*j+1}$

There are 20 constants A(r,i). They are equal to the B_i but are in reverse order, that is:
For $0 \le r \le 4$ and $1 \le i \le 4$
    Set $A(r,i) = B_{33-(4*r+i)}$

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 returns 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:

- $ROTL^n(Z)$=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)⊕(¬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. In hex we have:

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 hex, these are:

J_8 = 0x88888888888888888888888888888888
J_4 = 0x44444444444444444444444444444444
J_2 = 0x22222222222222222222222222222222
J_1 = 0x11111111111111111111111111111111

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 i from 0 to 3

Set $W_i = ROTL^{57}(SB([W_i + F(W_{i-1}) + Ch(W_{i-1},W_{i-2},W_{i-3}) + A(r,i)] \text{ modulo } 2^{128}))$

Set $(W_0, W_1, W_2, W_3) = BitMix(W_0, W_1, W_2, W_3)$

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 four and the computations of W_i are assumed to be iterative, so when each value is updated the updated value is used to update subsequent values. As mentioned the BitMix function operates on the words in parallel.

Message Schedule

The message schedule receives a 512 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 40 128 bit words.

For i from 8 to 32

Set $d_i = ROTL^{59}(SB([d_{i-8} + G(d_{i-1}) + Ch(d_{i-1},d_{i-2},d_{i-3}) + d_{i-4} + B_i] \text{ modulo } 2^{64}))$

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.

MS(0,D)=
BitMix($ROTL^{37}(d_0) \oplus d_4$, $ROTL^{37}(d_1) \oplus d_5$, $ROTL^{37}(d_2) \oplus d_6$, $ROTL^{37}(d_3) \oplus d_7$)
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_{48}. In hex, 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
```

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 $0 \le r \le 4$ and $1 \le i \le 4$

Set $A(r,i) = B_{33-(4*r+i)}$

SANDstorm Compression Function Performance

SANDstorm uses multiplication as one of the primary mixing agents. On most modern machines this operation is relatively 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 operates independent of the state variables. That means a large fraction of the work in the compression function may be accomplished in parallel and/or pipelined.

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.

6. Cutdown and Extension Alternatives

We were asked 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 method to cut down would be to chop of a 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 loose the resistance to multicollisions, herding and the like.

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.

It would not be in the spirit of the algorithm to change the superblock sizes. We don't feel that there is a security risk in changing the superblock sizes. However, we have not analyzed to any great extent the effect of changing the superblocks sizes on the fly or during execution, nor have we compared outputs of different size choices with different messages and the like. In this stage of the selection process, leaving the block sizes alone would be more appropriate.

We, of course, 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: Increase the number of rounds. Continue with the message schedule with the one step then 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. So the adjustment is not simple.

Option 2: 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 hash output. Therefore processing that information further may be a simple and straightforward 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, just repeat the For loop and the BitMix. Since all outputs are combined and eventually fed into Level 4, the finishing step provides additional strength.

7. Design Choices

In this section we discuss the primary design choices of the SANDstorm family. Several of these have been discussed above.

- The SANDstorm constants were chosen to be those either used by SHA-256 or derived in a similar fashion. The primary reason for this was to save space in situations where both SHA and SANDstorm might be simultaneously implemented. A secondary reason is that of back doors. The constants exist and they and their method of generation are 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.
- The multiplication is the workhorse of the mixing operations. For efficiency and bit mixing sake we modified the square. The function Z^2 mod 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 down side is that the square has the property 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 sake. Our sbox operation actually stores x $\oplus$ sbox(x) wo that we can, with a single xor, swap out x for sbox(x). Any other byte position requires other manipulations with the data to accomplish. Further, the application of the sbox is not our primary mixing operation, it is there to break up changes that happen to land in the low order byte position. To propagate small changes one must repeatedly dodge the low order byte.
- The BitMix function was chosen as a method to break up the algebraic dependencies that might appear in the round function and give further separation between what comes in via the message schedule. The BitMix function take influence from each state word and efficiently packs it into each byte.
- The mode was designed specifically so that parallelization could happen and the structure of the chaining values between rounds so that careful management may allow partial parallelization and pipelining within the round function.

8. Security Discussions

Collisions in the Chaining Values

It is a straight forward exercise 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 had 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 string of data blocks with a single block changed then the chaining values cannot collide. In a given superblock suppose D and D' are at block position j and suppose that the preceding data blocks are the same. Then the chaining values coming into position j must be the same. Now suppose that the chaining values moving into position j+1 are equal. Then

Starting at the bottom suppose that chn(4,j) = chn'(4,j) and that chn(3,j) = chn'(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. Sor for Round 4 we have that $chn(3,j) \oplus MS(4, D) = chn'(3,j) \oplus MS(4,D')$, and so MS(4,D)=MS(4,D'). Similarly by equating chn(2,j) and chn'(2,j) we determine MS(3,D)=MS(3,D'). From above this means that D = D'. This means that to construct a collision in the chaining values, one must manipulate at least two different data blocks. Of course this may not necessarily be true when the chaining values are combined to be the superblock output value to be moved into the next Level.

Message Schedule

We have tested the mixing effectiveness of the message schedule. Given an input data block $D=d_0, \ldots, d_7$ the message schedule is a powerful mixing operation. In particular, each bit of word d_{12} is a fairly strong function of each bit of $d_0, \ldots, d_7$ except d_3. The word d_3 enters the d_{12} as a sum simple sum of the other words hit with the sbox and rotated. Deltas in d_3 are passed directly into G(Z) in the computation of d_{12}. There are a couple of weak bits, namely those d_3 bits rotate into bit positions 60-63. By weak bits we mean 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 bits 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} is a strong function of each bit of $d_i, \ldots, d_{i+7}$. That means that if we take any group of contiguous eight words in the message schedule and then step our message schedule five times, the sixth word will be a full mix of the message state we started with.

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 function each bit of D. Similarly, each bit of
MS(2,D)=(d_{19}, d_{20}, d_{21}, d_{22}) is a function of each bit of ($d_6, \ldots, d_{13}$)
MS(3,D)=(d_{24}, d_{25}, d_{27}, d_{27}) is a function of each bit of ($d_{11}, \ldots, d_{18}$)
MS(4,D)=(d_{29}, d_{30}, d_{31}, d_{32}) is a function of each bit of ($d_{16}, \ldots, d_{23}$)

Note that SANDstorm steps and extra time so that MS(1, D) begins with d_{14}. This was done to provide additional distance between MS(0,D) which begins the state operations in the round functions.

Round Function

The round functions is not quite as complex as the message schedule and so does not mix quite as well as. Plus 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 le (W''_0, W''_1, W''_2, W''_3)=BitMix(W'_0, W'_1, W'_2, W'_3). We have that W'_0 is not a strong function of all of the W_i but successively the strength increases until each bit of W'_3 is a strong function of each bit of the input words, W_i. After the BitMix operation the each byte 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 the outputs a strong function of each input bit one round up. 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 are a full mix of the data two rounds previous.

Attacks on the Merkle-Damgard Structure

For a really long message, Level 3 acts like a typical Merkle-Damgard construction so it will be susceptible to the ills of that method. That is, long messages 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. That means that if some sort of random process is used to construct nefarious intermediate state, then those attacks will be completely foiled.

Less state is carried from level to level than from block position to block position. That means random attacks on the block data formed from superblock chaining would be more efficient. However, twice as much state is contained in the data blocks than is in the final

message digest. So, again, if random processes are used to construct nefarious data blocks, the work factor will be impractical for those attacks as well.

9. 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.

Do to their method of construction the timings for SANDstorm-224 and SANDstorm-256 will be virtually identical. Similarly for SANDstorm-512 and -384, so our timing results reflect only for SANDstorm-256 and SANDstorm-512.

An optimized version of SHA-1 and SHA-256 were used as reference points of comparison. The timings below focus on the time to complete a single compression function. This is where we spend most of our time optimizing our algorithms. Below are timings including our mode with various block sizes.

The compiler listed does not support assembly insertions, nor does it appear that assembly coded versions of the algorithm will be given much priority in Round 1 of the competition. However, we include the timing numbers as a reference.

	32-bit Machine		64-bit Machine	
	Optimized	Assembly	Optimized	Assembly
SANDstorm-256	1917 ns	1437 ns	639 ns	479 ns
SHA-1	2566 ns		2566 ns	
SHA-256	3218 ns		3218 ns	
SANDstrom-512	7668 ns		2556 ns	
SHA-512	9648 ns		3216 ns	

Figure 7: Relative Timings

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, at this time. We assume that the same relative speed enhancements would be available.

10. Memory Usage

SANDstorm-256 uses 50 of the 64 32-bit constants used by the SHA family, during the compression operation, it also uses the same eight initialization constants, and it also requires eight additional fixed constants. Two of the additional constants are 32 bits each and 6 are 64 bits. This is a total of 50*32 +8*32+448= 2304 bits. Only 448 constant bits are separate from what is needed for an implementation of SHA-256. Of those 448 bits we have that 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.

The message schedule computes 25 additional 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 we have 33*64=2112 bits. On the other hand the message schedule may be thought of as a block of 8 64 bit words and processed in an as-needed fashion. In this case there are only 512 bits to store.

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, conserving on the total memory usage. Total 2048 bits.

In the compression function, there are five rounds, each requiring a chaining variable that is 256 bits in length. (One of the chaining variables is actually a constant, determined after Level 0). Each of the five levels in the tree requires five chaining values. 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. So at any given time at most three of levels require storage of the chaining values. That is 5*3*256= 3840 bits. However, the chaining variables the initialization constants are tied together. The eight 32 bit initialization constants are expanded into five 256 bit initial chaining values. So, an implementation may, for simplicity sake, keep these values around for each level rather than reconstitute them as needed. So it is reasonable to expect to see an implementation require 5*5*256=6400 bits of chaining variable to be available rather than the 3840 that is minimally required.

Data is also passed to from one level of the tree to the next. At most this will require 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, there is a total of 1024 bits required to be passed from level to level.

At any given time the round function actively operates on four 64 bit state words requiring 256 bits.

	Constant	Volatile	Active	
Constants	2304			
AES sbox	2048			
Message Schedule		512		
Chaining Variables		3840-6400		
Level Data		1024		
State Words			256	
Totals	4352	5376-7936	256	9984-12544

Figure 8: Memory Usage

Note that for short messages the memory usage is reduced. For one block messages there is no level data aside from Level 0 and Level 4 in which Level 0 must be completed before Level 4 begins. At any given time only one set of five chaining values needs to be retained. This requires then 4352+512+5*256+256= 4352+2048=6400 bits.

Similarly, messages of shorter length will not require 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 will require this same amount of storage 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.

Note these calculations do not include memory usage for the code either compiled or not.

Appendix A Source Code for SANDstorm-256

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>

/* Type definitions */
typedef unsigned long long ULL;
typedef unsigned int UI;
typedef unsigned char BitSequence;
typedef unsigned long long DataLength; //typical 64 bit value

typedef enum {
      SUCCESS = 0,
      FAIL = 1,
      BAD_HASHBITLEN = 2
} HashReturn;

typedef struct {
      int blockIters[3];                  // iterators to manage which level we are at in the algorythm
      int hashbitlen;                     // length of the resultant hash
      int pipedBits;                      // current number of bits in the pipe to be processed
      int initCompressFlag;               // lets program know if we have accomplished the first compress
      BitSequence queuedData[64];         // data that are left over from last call to update
      DataLength prevBlock[3][5][4];      // data to feed into the next compression round
} hashState;

HashReturn Init(hashState *state, int hashbitlen);
HashReturn Update(hashState *state, const BitSequence *data, DataLength databitlen);
HashReturn Final(hashState *state, BitSequence *hashval);
HashReturn Hash(int hashbitlen, const BitSequence *data, DataLength databitlen, BitSequence *hashval);

#define     MAXBITS     512         /* Maximum number of bits for one run of the compression function */
#define     MAXBYTES    64          /* Maximum number of bytes for one run of the compression function */
#define     BYTELENGTH  8
```

```
#define L1SBLENGTH      10
#define L2SBLENGTH      100

/* r and s, used in G(x) function */
#define RCONST 0xA611186BLL
#define SCONST 0xBEE8390DLL

/* used to select the high or low 32 bits of a 64-bit variable */
#define HIWORD_MASK 0xffffffff00000000LL
#define LOWORD_MASK 0xffffffffLL

static const BitSequence WORDMASK = 0xFF;

typedef unsigned long long ULL;
typedef unsigned int UI;

ULL rand64(void);
void seedrand64(int seed);
void toBin64(ULL word, char binstring[65]);
ULL fromBin(const char * binstring);

/* Constants as specified in the write up document for SANDstorm */
DataLength levelOneToThreeConst[3][5][4];
DataLength static const mainConstantInputWords[5][4] = {
      {0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull},
      {0xbb67ae853c6ef372ull, 0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull},
      {0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull},
      {0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull, 0xbb67ae853c6ef372ull},
      {0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull}
};
DataLength levelFourConstants[5][4] = {
      {0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull},
      {0xbb67ae853c6ef372ull, 0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull},
      {0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull},
      {0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull, 0xbb67ae853c6ef372ull},
      {0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull}
};
```

```
DataLength initialVectorTempWords[5][4] = {
      {0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull},
      {0xbb67ae853c6ef372ull, 0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull},
      {0x3c6ef372a54ff53aull, 0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull},
      {0xa54ff53a510e527full, 0x9b05688c1f83d9abull, 0x5be0cd196a09e667ull, 0xbb67ae853c6ef372ull},
      {0x510e527f9b05688cull, 0x1f83d9ab5be0cd19ull, 0x6a09e667bb67ae85ull, 0x3c6ef372a54ff53aull}
};

/* C_i used in the message schedule function */
static const ULL c_const[32] = {
      4467991496273718166ULL, 10320099885548846075ULL, 3652204158256733020ULL, 151713105414105658ULL,
      1809822736342558228ULL, 8491746423542996155ULL, 12148758605418197427ULL, 427832072915204837ULL,
      1455895431583153478lULL, 2243670746741826757ULL, 14736213068173458911ULL, 4283033351706511836ULL,
      1230965383734654956 3ULL, 3732889600109516124ULL, 13506051301407686133ULL, 855994113871575081ULL,
      1791647498905438856ULL, 10966890579359513908ULL, 13573654079133674501ULL, 12431660292823995810ULL,
      1372266757111390945 9ULL, 3932965942609372318ULL, 5706647085361811113ULL, 6082198129822485 20ULL,
      1109989185458561604ULL, 12419788189904350619ULL, 667347512028540877ULL, 11782682359274698311ULL,
      1601089142644776575 0ULL, 2315841336524277508ULL, 17648493929504081603ULL, 4469244204797080503ULL};

/* B_i used in the round function */
static const ULL B[5][4] = {
      {7477780175313128172ULL, 3012268973459497798ULL, 8989841230781802381ULL, 8528179847982205338ULL},
      {7363963421162251098ULL, 2692313750071500500ULL, 1237217737923967899 3ULL, 1045525886582774919ULL},
      {8136103064137497703ULL, 4467991496273718166ULL, 1032009988554884607 5ULL, 3652204158256733020ULL},
      {1517131054141056 58ULL, 1809822736342558228ULL, 8491746423542996155ULL, 1214875860541819742 7ULL},
      {4278320729152048 37ULL, 1455895431583153478 1ULL, 2243670746741826757ULL, 1473621306817345891ULL}};

/* SB(x) = x ^ AES_sbox[x] */
static const unsigned char fsbox[256] = {
      99, 125, 117, 120, 246, 110, 105, 194, 56, 8, 109, 32, 242, 218, 165, 121,
      218, 147, 219, 110, 238, 76, 81, 231, 181, 205, 184, 180, 128, 185, 108, 223,
      151, 220, 177, 5, 18, 26, 209, 235, 28, 140, 207, 218, 93, 245, 31, 58,
      52, 246, 17, 240, 44, 163, 51, 173, 63, 43, 186, 217, 215, 26, 140, 74,
      73, 194, 110, 89, 95, 43, 28, 231, 26, 114, 156, 248, 101, 174, 97, 203,
      3, 128, 82, 190, 116, 169, 231, 12, 50, 146, 228, 98, 22, 17, 6, 144,
      176, 142, 200, 152, 39, 40, 85, 226, 45, 144, 104, 20, 60, 81, 241, 199,
      33, 210, 50, 252, 230, 232, 78, 130, 196, 207, 160, 90, 108, 130, 141, 173,
      77, 141, 145, 111, 219, 18, 194, 144, 76, 46, 244, 182, 232, 208, 151, 252,
```

```
      240, 16, 221, 79, 182, 191, 6, 31, 222, 119, 34, 143, 66, 195, 149, 68,
      64, 147, 152, 169, 237, 163, 130, 251, 106, 122, 6, 201, 61, 56, 74, 214,
      87, 121, 133, 222, 57, 96, 248, 30, 212, 239, 78, 81, 217, 199, 16, 183,
      122, 185, 231, 237, 216, 99, 114, 1, 32, 20, 190, 212, 135, 112, 69, 69,
      160, 239, 103, 181, 156, 214, 32, 217, 185, 236, 141, 98, 90, 28, 195, 65,
      1, 25, 122, 242, 141, 60, 104, 115, 115, 247, 109, 2, 34, 184, 198, 48,
      124, 80, 123, 254, 75, 19, 180, 159, 185, 96, 215, 244, 76, 169, 69, 233};

/* Offset for message schedule inputs, used in round function */
static const int first_m[5] = {4, 12, 20, 28, 36};

/* Number of bits to rotate in message schedule and in round functions */
#define MS_ROT_BITS     51
#define R_ROT_BITS      19

/*===== Global variables used in ModMix =====*/
ULL MS[40]; /* message schedule buffer*/
ULL w[4]; /* round buffer */

void createMessageFromBlock(DataLength M[8], DataLength input[5][4]);
unsigned short Do_Block_ModMix_Ref (ULL db[8], ULL block_buffer[5][4]);
void compress(hashState *state);
void initCompress(hashState *state);
ULL F(ULL x);
ULL G(ULL x);
ULL Ch(ULL a, ULL b, ULL c);
ULL RotLeft(ULL n, int i);
ULL Rot32(ULL n);
ULL SB(ULL x);
ULL C(int i);
int mm(int i);
ULL Msg_M(int i);
void round_in(ULL buffer[4], int);
void round_out(ULL prev_block[4], ULL current_block[4]);
void round4_out(ULL current_block[4]);
void do_round(int round_num);
void block_init();
```

```
/*
 * Initializes a hashState with the intended hash length of this particular
 * instantiation. Additionally, any data independent setup is performed.
 * Parameters:
 *          state: a structure that holds the hashState information
 *          hashbitlen: an integer value that indicates the length of the hash output in bits.
 * Returns:
 *          SUCCESS - on success
 *          BAD_HASHBITLEN - hashbitlen is invalid (e.g., unknown Value)
 */
HashReturn Init(hashState *state, int hashBitLen){
      int i,j,k;

      state->hashbitlen = hashBitLen;
      state->pipedBits = 0;
      state->initCompressFlag = 0;
      state->blockIters[0] = 0;
      state->blockIters[1] = 0;
      state->blockIters[2] = 0;
}

/*
 * Process the supplied data.
 * Parameters:
 *          state: a structure that holds the hashState information
 *          data: the data to be hashed
 *          databitlen: the length, in bits, of the data to be hashed
 * Returns:
 *          SUCCESS - on success
 *          HashReturn Update(hashState *state, const BitSequence *data,
 *                DataLength databitlen);
 */
HashReturn Update(hashState *state, const BitSequence *data, DataLength databitlen){
      int neededBits, qDataIndex, bytesToCpy,i;
      ULL dataBitsLeft,dataIndex;

      dataIndex = 0;
      dataBitsLeft = databitlen;
```

```
qDataIndex = state->pipedBits / BYTELENGTH;

// byteOffset = state->pipedBits >> 3;
neededBits = MAXBITS - state->pipedBits;

// This will check to see if we have enough bits to run compress (512), and if not it
// will add those bits to the queue and then return
if(dataBitsLeft < neededBits ){
      // if we are on the last call to update we could have a bits left that is not divisible
      // by 8. So if we have any bits we have to make sure we get the full last byte.
      if( (dataBitsLeft & 7) > 0 ){
            bytesToCpy = (databitlen / BYTELENGTH) + 1;
      }else{
            bytesToCpy = databitlen / BYTELENGTH;
      }
      memcpy(&(state->queuedData[qDataIndex]),data,(size_t)bytesToCpy);
      state->pipedBits += (int)databitlen;
      return SUCCESS;
}

//    This is true when the piped bits are greater than 0, but not yet 512, but because
//    we passed the previous condition we know we have enough bits to process between
//    what is in the queue already and what has currently been input to the function
if(state->pipedBits > 0){
      // figure out how many bytes are needed to complete 512 bits then put them into
      // the message queue
      bytesToCpy = neededBits / BYTELENGTH;
      memcpy(&(state->queuedData[qDataIndex]),data,(size_t)bytesToCpy);
      state->pipedBits = MAXBITS;
      dataIndex = bytesToCpy;
      dataBitsLeft -= neededBits;

      // now we need to check if this is the first call to compress, and if it is it will
      // require a special first run to set state variables to all subsequent calls to compress.
      // if it is not the first time then do a normal compress

      if(state->initCompressFlag == 0){
            initCompress(state);
```

```
		}else{
			compress(state);
		}
	}

	while(dataBitsLeft>=MAXBITS){
		memcpy(state->queuedData,&data[dataIndex],MAXBYTES);
		dataBitsLeft -= MAXBITS;
		dataIndex += MAXBYTES;
		state->pipedBits = MAXBITS;

		if(state->initCompressFlag == 0){
			initCompress(state);
		}else{
			compress(state);
		}
	}

	state->pipedBits = (int)dataBitsLeft;
	if( (dataBitsLeft & 7) > 0 ){
		bytesToCpy = (dataBitsLeft / BYTELENGTH) + 1;
	}else{
		bytesToCpy = dataBitsLeft / BYTELENGTH;
	}
	memcpy(state->queuedData,&data[dataIndex],(size_t)bytesToCpy);
	return SUCCESS;
}

/*
 * Perform any post processing and output filtering required and return the final hash value.
 * Parameters:
 *		state: a structure that holds the hashState information
 *		hashval: the storage for the final hash value to be returned
 * Returns:
 *		SUCCESS - on success
 *		HashReturn Final(hashState *state, BitSequence *hashval);
 */
HashReturn Final(hashState *state, BitSequence *hashval){
```

```
int lastDataByteIndex, validBits, i,j;
BitSequence mask, str[8], ormask;
DataLength Mess[8], messageLength;

// if we have some bits leftover, pad them with a one bit value of 1 and then zeros till 512 bits
validBits = state->pipedBits & 7;
lastDataByteIndex = state->pipedBits >> 3;

if(validBits > 0){
      // last byte index will be the index of the byte that we have an integral number of bits
      // left to deal with. i.e. we have 4 bits left on the character indexed by this value.
      mask = WORDMASK << (7-validBits);

      // this mask will zero out the non message bits.
      state->queuedData[lastDataByteIndex] = state->queuedData[lastDataByteIndex] & mask;

      // set the bit to one to start the padding then perform inclusive or on last character
      ormask = ORMASK << (7-validBits);
      state->queuedData[lastDataByteIndex] = state->queuedData[lastDataByteIndex] | ormask;
}else{
      state->queuedData[lastDataByteIndex] = 0x80;
}

// pad the rest of the message out with zeros
for(i=lastDataByteIndex+1;i<64;i++){
      state->queuedData[i] = 0x00;
}

// If we haven't done Level 0 compress do it, and go straight to Level 4 from there
if(state->initCompressFlag == 0){
      // no level zero has been performed so we run here and set the message for level 4
      initCompress(state);
      createMessageFromBlock(Mess,initialVectorTempWords);
      messageLength = state->pipedBits;
}else{
      messageLength = (state->blockIters[0] + 1) * MAXBITS + state->pipedBits; // add 1 for level 0
      // perform one last run of compress then find out where last ouput is at
      compress(state);
```

```
			createMessageFromBlock(Mess,state->prevBlock[0]);
			// handle last set of do_block calls
			if(state->blockIters[1] > 0){
				Do_Block_ModMix_Ref(Mess,state->prevBlock[1]);
				createMessageFromBlock(Mess,state->prevBlock[1]);
				if(state->blockIters[2] > 0){
					Do_Block_ModMix_Ref(Mess,state->prevBlock[2]);
					createMessageFromBlock(Mess,state->prevBlock[2]);
				}
			}
		}

		// xor message length into level four constants
		for(i=0;i<5;i++){
			for(j=0;j<4;j++){
				levelFourConstants[i][j] = levelFourConstants[i][j] ^ messageLength;
			}
		}

		Do_Block_ModMix_Ref(Mess,levelFourConstants);

		createMessageFromBlock(Mess,levelFourConstants);

		// Final hash is in Mess so move them into hasval
		for(j=0;j<8;j++){
			BitSequence tmp;
			for(i=7;i>=0;i--){
				int finalValIndex = (j*8) + (7-i);
				hashval[finalValIndex] = (char)(Mess[j] >> (i*8));
			}
		}
		return SUCCESS;
}

/*
 * Hash the supplied data and provide the resulting hash value. Set return code as
 * appropriate.
 * Parameters:
```

```
 *          hashbitlen: the length in bits of the desired hash value
 *          data: the data to be hashed
 *          databitlen: the length, in bits, of the data to be hashed
 *          hashval: the resulting hash value of the provided data
 * Returns:
 *          SUCCESS - on success
 *          FAIL - arbitrary failure
 *          BAD_HASHBITLEN - unknown hashbitlen requested
 *          ...
 */
HashReturn Hash(int hashbitlen, const BitSequence data[], DataLength databitlen, BitSequence *hashval){
      hashState state;

      // First initialize
      Init(&state,hashbitlen);

      // Hash entire message
      Update (&state, data, databitlen);

      // Perform finalize passing the var hashval to hold the data
      Final(&state,hashval);

      // return status code
}

void initCompress(hashState *state){
      int i, j, k;
      DataLength Mess[8];
      state->initCompressFlag = 1;

      // Copy the data in the queue to a variable capable of holding it as 8 64 bit words (ULL)
      for(i=0;i<8;i++){
            memcpy(&Mess[i],&(state->queuedData[i*8]),8);
      }

      // perform initial do_block and store all outputs into the array LevelZeroConst, we won't need
      // these any more for the rest of the algorithm
      Do_Block_ModMix_Ref(Mess, initialVectorTempWords);
```

```
// now we need to set up the constants for levels 1 through 3 constants as some function of the
// output of the initial block index works like for C0 words it is mainConstINW[0][i] C1=MCIW[1][i]
// for level 1
for(i=0;i<4;i++){
	levelOneToThreeConst[0][0][i] = mainConstantInputWords[0][i] ^ initialVectorTempWords[4][i];
	//for level 1 constant 0 = co ^ s4
	levelOneToThreeConst[0][1][i] = mainConstantInputWords[1][i] ^ initialVectorTempWords[1][i];
	//for level 1 constant 1 = c1 ^ s1
	levelOneToThreeConst[0][2][i] = mainConstantInputWords[2][i] ^ initialVectorTempWords[2][i];
	//for level 1 constant 2 = c2 ^ s2
	levelOneToThreeConst[0][3][i] = mainConstantInputWords[3][i] ^ initialVectorTempWords[3][i];
	//for level 1 constant 3 = c3 ^ s3
	levelOneToThreeConst[0][4][i] = mainConstantInputWords[4][i] ^ initialVectorTempWords[4][i];
	//for level 1 constant 4 = c4 ^ s4
}

// level 2
for(i=0;i<5;i++){
	for(j=0;j<4;j++){
		// level 2
		levelOneToThreeConst[1][i][j] = ~(levelOneToThreeConst[0][i][j]);
		// level 3
		levelOneToThreeConst[2][i][j] = levelOneToThreeConst[0][i][j];
		// level 4
		levelFourConstants[i][j] = ~(mainConstantInputWords[i][j]);
	}
}

// Copy over new constants into the prevBlock array for subsequent calls to compress
for(i=0;i<3;i++){
	for(j=0;j<5;j++){
		for(k=0;k<4;k++){
			state->prevBlock[i][j][k] = levelOneToThreeConst[i][j][k];
		}
	}
}
```

```
void compress(hashState *state){
	// First thing to do is convert the BitSequence character array to 8 64 bit ULLs
	int i, j;
	DataLength MessL1[8],MessL2[8],MessL3[8],levelOneSuperBlock, levelTwoSuperBlock;

	// Copy the data in the queue to a variable capable of holding it as 8 64 bit words (ULL)
	for(i=0;i<8;i++){
		memcpy(&MessL1[i],&(state->queuedData[i*8]),8);
	}

	/* This will evaluate to true when there is enough information to run a compress
	 * at the last level of mod mix. If we have 100 level 1 blocks complete then we
	 * pass the outputs of rounds 4 and 5 up to the third level as the message to a
	 * mod mix block.
	 */
	if(((state->blockIters[1]-1)%100) == 0 && (state->blockIters[1] > 1) ){
		createMessageFromBlock(MessL3,state->prevBlock[1]);
		// run block function on level 3
		Do_Block_ModMix_Ref(MessL3,state->prevBlock[2]);
		// Reset rounds 2 through 5 for the next set of 100
		levelTwoSuperBlock = state->blockIters[1] / L2SBLENGTH;
		for(i=0;i<4;i++){
			for(j=0;j<4;j++){
				state->prevBlock[1][i+1][j] = levelOneToThreeConst[1][i+1][j] ^ levelTwoSuperBlock;
			}
		}
	}
	if(((state->blockIters[0]-1)%10) == 0 && state->blockIters[0] > 1){
		createMessageFromBlock(MessL2,state->prevBlock[0]);
		// run block function on the 100 block level
		Do_Block_ModMix_Ref(MessL2,state->prevBlock[1]);
		// Reset rounds 2 through 5 for the next set of 10 and xor the super block #
		levelOneSuperBlock = state->blockIters[0] / L1SBLENGTH;
		for(i=0;i<4;i++){
			for(j=0;j<4;j++){
				state->prevBlock[0][i+1][j] = levelOneToThreeConst[0][i+1][j] ^ levelOneSuperBlock;
			}
		}
```

```
            // increment the block counters
            state->blockIters[1]++;
      }
      // run block function on the 10 blcok level
      Do_Block_ModMix_Ref(MessL1,state->prevBlock[0]);
      state->blockIters[0]++;
}

void createMessageFromBlock(DataLength M[8], DataLength input[5][4]){
      M[0] = input[1][0] ^ input[3][0];
      M[1] = input[1][1] ^ input[3][1];
      M[2] = input[1][2] ^ input[3][2];
      M[3] = input[1][3] ^ input[3][3];
      M[4] = input[2][0] ^ input[4][0];
      M[5] = input[2][1] ^ input[4][1];
      M[6] = input[2][2] ^ input[4][2];
      M[7] = input[2][3] ^ input[4][3];
}

/*
 *  Name:  Do_Block_ModMix_Ref - This function hashes a 512-bit block using the ModMix Hash Algorithm
 */
unsigned short Do_Block_ModMix_Ref (ULL data_input[8], ULL prevBlockArr[5][4]) {
      unsigned short i;
      unsigned short ercode = NO_HASH_ERROR; /* initialize error return flag */

      /*===== MESSAGE SCHEDULE =====*/
      /* First 8 M's of the message schedule are the input into this function */
      for (i=0; i < 8; i++)
            MS[i] = data_input[i];

      /* Calculate the rest of the message schedule */
      for (i=8; i < 40; i++)
            MS[i] = Msg_M(i);

      /* Set up inputs for the first round; this is an XOR of the data input. */
      for (i=0; i < 4; i++)
            MS[i+4] = MS[i] ^ MS[i+4];
```

```
	/*===== ROUND FUNCTIONS =====*/
	// Round 0
	round_in(prevBlockArr[0], first_m[0]);
	do_round(0);
	round_out(prevBlockArr[1], prevBlockArr[1]);

	// Round 1
	round_in(prevBlockArr[1], first_m[1]);
	do_round(1);
	round_out(prevBlockArr[2],prevBlockArr[1]);

	// Round 2
	round_in(prevBlockArr[1], first_m[2]);
	do_round(2);
	round_out(prevBlockArr[3],prevBlockArr[2]);

	// Round 3
	round_in(prevBlockArr[2], first_m[3]);
	do_round(3);
	round_out(prevBlockArr[4],prevBlockArr[3]);

	// Round 4
	round_in(prevBlockArr[3], first_m[4]);
	do_round(4);
	round4_out(prevBlockArr[4]);

	return ercode;
}

/* Set the round buffer "w" for the next round. */
/* NOTE: In rounds 0-2, w is XOR'ed with a constant; in rounds 3-4, w is XOR'ed with the state of the
previous block (unless it's the first block in a section, in which case "the state of the previous block"
will be a constant */
void round_in(ULL buffer[4], int ms_index) {
	w[0] = buffer[0] ^ MS[ms_index];
	w[1] = buffer[1] ^ MS[ms_index + 1];
	w[2] = buffer[2] ^ MS[ms_index + 2];
```

```
	w[3] = buffer[3] ^ MS[ms_index + 3];
}

/* Once the round buffer has been set, do the round. Round_num tells the function which M's to get from the
message schedule */
void do_round(int round_num) {
	w[0] = RotLeft(SB(w[0]+F(w[3])+Ch(w[3],w[2],w[1])+B[round_num][0]),R_ROT_BITS);
	w[1] = RotLeft(SB(w[1]+F(w[0])+Ch(w[0],w[3],w[2])+B[round_num][1]),R_ROT_BITS);
	w[2] = RotLeft(SB(w[2]+F(w[1])+Ch(w[1],w[0],w[3])+B[round_num][2]),R_ROT_BITS);
	w[3] = RotLeft(SB(w[3]+F(w[2])+Ch(w[2],w[1],w[0])+B[round_num][3]),R_ROT_BITS);
}

/* Sets the given buffer equal to the current state of the round buffer */
void round_out(ULL prev_block[4], ULL current_block[4]) {
	current_block[0] = prev_block[0] ^ w[0];
	current_block[1] = prev_block[1] ^ w[1];
	current_block[2] = prev_block[2] ^ w[2];
	current_block[3] = prev_block[3] ^ w[3];
}

/* Sets the given buffer equal to the current state of the round buffer, after XORing it with buffer_in */
void round4_out(ULL current_block[4]) {
	current_block[0] = w[0];
	current_block[1] = w[1];
	current_block[2] = w[2];
	current_block[3] = w[3];
}

/* Rotate a 64-bit variable left by i bits */
ULL RotLeft(ULL n, int i) {
	return (n << i) | (n >> (64 - i));
}

/* Rotate a 64-bit variable 32 bits.  Left or Right makes no difference. */
ULL Rot32(ULL n) {
	return (n << 32) | (n >> (32));
}
```

```
/* F(x) used in round function */
ULL F(ULL x) {
      ULL y, z, f_res;

      y = x >> 32;
      z = x & LOWORD_MASK;
      f_res = y*y + z*z;
      return f_res;
}

/* F(x) used in message schedule */
ULL G(ULL x) {
      ULL y, z, g_res;

      y = x >> 32;
      z = x & LOWORD_MASK;
      g_res = (y*y + z*z + Rot32(((y + RCONST) & 0xffffffff)*((z + SCONST) & 0xffffffff)));
      return g_res;
}

/* Choose function, used in both round function and message schedule */
ULL Ch(ULL a, ULL b, ULL c) {
      return (a & b) ^ (~a & c);
}

/* SB(x), used in both round function and message schedule.  Replaces low byte with member of AES sbox
array. */
ULL SB(ULL x) {
      return (x ^ fsbox[(unsigned char)(x & 0xff)]);
}

/* This is the main message schedule function */
ULL Msg_M(int i) {
      return RotLeft(SB(MS[i-8]+c_const[i-8]+G(MS[i-1])+Ch(MS[i-1],MS[i-2],MS[i-3])+MS[i-4]),MS_ROT_BITS);
}
```

Appendix B Source Code for SANDstorm-512

```
// 512 bit hash function
// uses arithmetic on 128-bit quantities
typedef unsigned long long u64;
typedef struct { u64 hi; u64 lo; } u128;

// make a 128 bit constant from two 64 bit constants
u128 k128(u64 hi, u64 lo) { u128 hl; hl.hi = hi; hl.lo = lo; return hl; }

// add two 128 bit numbers, dropping any carry
u128 a128(u128 x, u128 y)
{ u128 sum;  sum.hi = x.hi + y.hi; sum.lo = x.lo + y.lo;
 if (sum.lo<x.lo) sum.hi++; return sum; }

// xor two 128 bit numbers
u128 x128(u128 x, u128 y)
{ u128 xor; xor.hi = x.hi ^ y.hi; xor.lo = x.lo ^ y.lo; return xor; }

// complement a 128 bit number
u128 c128(u128 x)
{ u128 comp; comp.hi = ~ x.hi; comp.lo = ~ x.lo; return comp; }

// and two 128 bit numbers
u128 b128(u128 x, u128 y)
{ u128 and; and.hi = x.hi & y.hi; and.lo = x.lo & y.lo; return and; }

// rotate left a 128 bit number
u128 r128(u128 x, int sh)
{ u128 rot; sh &= 127;
 if (sh==0)       { rot.hi = x.hi; rot.lo = x.lo; }
 else if (sh<64)  { rot.hi = (x.hi << sh) | (x.lo >> (64-sh));
                    rot.lo = (x.lo << sh) | (x.hi >> (64-sh)); }
 else if (sh==64) { rot.hi = x.lo; rot.lo = x.hi; }
 else             { rot.hi = (x.lo << (sh-64)) | (x.hi >> (128-sh));
                    rot.lo = (x.hi << (sh-64)) | (x.lo >> (128-sh)); };
 return rot; }
```

```
#define msk32 0xFFFFFFFFul
#define b32 0x100000000ull

// multiply two 64 bit numbers, producing a 128 bit product
u128 m128(u64 x, u64 y)
{ u128 prod;
  u64 xhi = x>>32, xlo = x&msk32, yhi = y>>32, ylo = y&msk32;
  u64 tmp,tmp2;
  prod.hi = xhi*yhi; prod.lo = xlo*ylo; tmp = xlo*yhi; tmp2 = xhi*ylo;
  tmp += tmp2; if (tmp<tmp2) prod.hi += b32;
  prod.hi += tmp>>32;
  tmp <<= 32;
  prod.lo += tmp; if (prod.lo<tmp) prod.hi++;
  return prod; }

u128 sandwf(u128 x)
{ return a128(m128(x.hi,x.hi),m128(x.lo,x.lo)); }

u64 sandwga = 0xb123456789abcdedull, sandwgb = 0xa21436587a9cbeddull;

u128 sandwg(u128 x)
{ return a128(m128(x.hi,x.hi),
              a128(m128(x.lo,x.lo),r128(m128(x.hi+sandwga,x.lo+sandwgb),64))); }

/* SB(x) = x ^ AES_sbox[x] */
static const unsigned char fsbox[256] =
{ 99, 125, 117, 120, 246, 110, 105, 194, 56, 8, 109, 32, 242, 218, 165, 121,
  218, 147, 219, 110, 238, 76, 81, 231, 181, 205, 184, 180, 128, 185, 108, 223,
  151, 220, 177, 5, 18, 26, 209, 235, 28, 140, 207, 218, 93, 245, 31, 58,
  52, 246, 17, 240, 44, 163, 51, 173, 63, 43, 186, 217, 215, 26, 140, 74,
  73, 194, 110, 89, 95, 43, 28, 231, 26, 114, 156, 248, 101, 174, 97, 203,
  3, 128, 82, 190, 116, 169, 231, 12, 50, 146, 228, 98, 22, 17, 6, 144,
  176, 142, 200, 152, 39, 40, 85, 226, 45, 144, 104, 20, 60, 81, 241, 199,
  33, 210, 50, 252, 230, 232, 78, 130, 196, 207, 160, 90, 108, 130, 141, 173,
  77, 141, 145, 111, 219, 18, 194, 144, 76, 46, 244, 182, 232, 208, 151, 252,
  240, 16, 221, 79, 182, 191, 6, 31, 222, 119, 34, 143, 66, 195, 149, 68,
  64, 147, 152, 169, 237, 163, 130, 251, 106, 122, 6, 201, 61, 56, 74, 214,
```

```
  87, 121, 133, 222, 57, 96, 248, 30, 212, 239, 78, 81, 217, 199, 16, 183,
  122, 185, 231, 237, 216, 99, 114, 1, 32, 20, 190, 212, 135, 112, 69, 69,
  160, 239, 103, 181, 156, 214, 32, 217, 185, 236, 141, 98, 90, 28, 195, 65,
  1, 25, 122, 242, 141, 60, 104, 115, 115, 247, 109, 2, 34, 184, 198, 48,
  124, 80, 123, 254, 75, 19, 180, 159, 185, 96, 215, 244, 76, 169, 69, 233};

// apply the AES sbox to the low byte of X
u128 sb128(u128 x)
{ u128 val; val.hi = x.hi; val.lo = x.lo ^ fsbox[x.lo & 255]; return val; }

// constants for message schedule & round constants
// these are a subset of the SHA2 round constants for SHA-512
// fractional part (in hex) of the cube roots of the first fifty primes
// the leading digits aren't completely random, but it's good enough
u128 sandwmsc[] = {
  0x428a2f98d728ae22ull, 0x7137449123ef65cdull, 0xb5c0fbcfec4d3b2full, 0xe9b5dba58189dbbcull,
  0x3956c25bf348b538ull, 0x59f111f1b605d019ull, 0x923f82a4af194f9bull, 0xab1c5ed5da6d8118ull,
  0xd807aa98a3030242ull, 0x12835b0145706fbeull, 0x243185be4ee4b28cull, 0x550c7dc3d5ffb4e2ull,
  0x72be5d74f27b896full, 0x80deb1fe3b1696b1ull, 0x9bdc06a725c71235ull, 0xc19bf174cf692694ull,
  0xe49b69c19ef14ad2ull, 0xefbe4786384f25e3ull, 0x0fc19dc68b8cd5b5ull, 0x240ca1cc77ac9c65ull,
  0x2de92c6f592b0275ull, 0x4a7484aa6ea6e483ull, 0x5cb0a9dcbd41fbd4ull, 0x76f988da831153b5ull,
  0x983e5152ee66dfabull, 0xa831c66d2db43210ull, 0xb00327c898fb213full, 0xbf597fc7beef0ee4ull,
  0xc6e00bf33da88fc2ull, 0xd5a79147930aa725ull, 0x06ca6351e003826full, 0x142929670a0e6e70ull,
  0x27b70a8546d22ffcull, 0x2e1b21385c26c926ull, 0x4d2c6dfc5ac42aedull, 0x53380d139d95b3dfull,
  0x650a73548baf63deull, 0x766a0abb3c77b2a8ull, 0x81c2c92e47edaee6ull, 0x92722c851482353bull,
  0xa2bfe8a14cf10364ull, 0xa81a664bbc423001ull, 0xc24b8b70d0f89791ull, 0xc76c51a30654be30ull,
  0xd192e819d6ef5218ull, 0xd69906245565a910ull, 0xf40e35855771202aull, 0x106aa07032bbd1b8ull,
  0x19a4c116b8d2d0c8ull, 0x1e376c085141ab53ull };

// where to start picking in Message Schedule
int sandwmspick[] = { 4, 14, 19, 24, 29 };

void mscpr(u128 *ms)
{ int i; printf("ms ");
   for (i=0;i<33;i++) { printf(" %016llx %016llx",ms[i].hi,ms[i].lo);
   if (i&1) printf("\n   "); } printf("\n"); }
```

```
// Compute the message schedule
// Assumes the message is present in the first 8 words (128 bits each)
// Begin by xoring the first 4 words (rotated) into the second 4 words.
// Then apply the mixing function to compute words after the first 8.
void sandwmsgsch(u128 *MS)
{ int i;
 for (i=8;i<33;i++)
  MS[i] = r128(sb128(a128(    MS[i-8],
                      a128(   sandwmsc[i-8],
                       a128(  sandwg(MS[i-1]),
                  a128( x128(b128(MS[i-1],MS[i-2]),   // Ch(i-1,i-2,i-3)
                                  b128(c128(MS[i-1]),MS[i-3])),
                             MS[i-4])))),
               57 /* was sandwmsr[i-8] */ );
 for (i=0;i<4;i++) MS[i+4] = x128(r128(MS[i],37),MS[i+4]); // mscpr(MS);
 }

const u128 ym8 = { 0x8888888888888888ull, 0x8888888888888888ull};
const u128 ym4 = { 0x4444444444444444ull, 0x4444444444444444ull};
const u128 ym2 = { 0x2222222222222222ull, 0x2222222222222222ull};
const u128 ym1 = { 0x1111111111111111ull, 0x1111111111111111ull};

// Run the compression function
// message (in *msg) is 8 128-bit words;
// prior block output is in prev[20*level+4*round+word]; result goes here too.
// inner state is w[0-3], 4 128-bit words
void sandwcmprs(u128 msg[], u128 prev[], int level, int xrnds)
{ int i,j,rnd; u128 MS[33], w[4];  u128 ym[4];  // yellowmix
 for (i=0;i<8;i++) MS[i] = msg[i]; sandwmsgsch(MS);
 for (i=0;i<4;i++) w[i] = prev[20*level+4+i];
 for (rnd=0;rnd<=4;rnd++)
 { for (i=0;i<4;i++) w[i] = x128(w[i],MS[sandwmspick[rnd]+i]);
   for (j=0;j<=((rnd<4)?0:xrnds);j++)
   {for (i=0;i<4;i++)
    w[i] = r128(sb128(a128(   w[i],
                      a128(  sandwf(w[(i+3)&3]),
                       a128( x128(b128(w[(i+3)&3],w[i^2]),
                          b128(c128(w[(i+3)&3]),w[(i+1)&3])),
```

```
                        sandwmsc[19-i-(rnd<<2)])))),
            57);
    // bitmix
    for (i=0;i<4;i++) ym[i] = x128(b128(w[i],ym8),
                          x128(b128(w[(i+1)&3],ym4),
                                      x128(b128(w[i^2],ym2),
                                           b128(w[(i+3)&3],ym1))));
    for (i=0;i<4;i++) w[i] = ym[i]; }
   if (rnd<4) for (i=0;i<4;i++) w[i] = x128(w[i],prev[20*level+4*rnd+4+i]);
   for (i=0;i<4;i++) prev[20*level+4*rnd+i] = w[i]; };
 }

typedef struct
{ unsigned char msgbytes[128];  // data bytes of the message
  int mbcnt;                    // number of message bytes in the buffer
  u128 msg[8];                  // message repacked into 8 words of 128 bits
  u128 msglen;                  // message length in bits
  u128 blkn1, blkn2;            // block numbers for level 1 and 2
  int lvl0flag;                 // level 0 has been processed;
  int lvl1cnt, lvl2cnt;         // number of blocks in current level 1 & 2 superblocks
  int lvl3flag;                 // level 3 has been used
  u128 prev[5*5*4];             // result of hash round function at each level
                                // 5 levels, 5 rounds, 4 128-bit words
  int outsize;                  // output size in bits; must be 384 or 512.
  u128 *lvlc;                   // pointer to level starting constants
 } sandwstate;

// constants for initializing (super)blocks at various levels
// based on the SHA-384/512 initial-value constants, which are
// derived from the fractional part (in hex) of the square roots
// of the primes 2-19 (for SHA-512) and 23-53 (for SHA-384).
u128 sandwlvlc384[20] =
{ 0xcbbb9d5dc1059ed8ull, 0x629a292a367cd507ull,
  0x9159015a3070dd17ull, 0x152fecd8f70e5939ull,
  0x67332667ffc00b31ull, 0x8eb44a8768581511ull,
  0xdb0c2e0d64f98fa7ull, 0x47b5481dbefa4fa4ull,

                         0x629a292a367cd507ull,
```

```
  0x9159015a3070dd17ull, 0x152fecd8f70e5939ull,
  0x67332667ffc00b31ull, 0x8eb44a8768581511ull,
  0xdb0c2e0d64f98fa7ull, 0x47b5481dbefa4fa4ull,
  0xcbbb9d5dc1059ed8ull,

  0x9159015a3070dd17ull, 0x152fecd8f70e5939ull,
  0x67332667ffc00b31ull, 0x8eb44a8768581511ull,
  0xdb0c2e0d64f98fa7ull, 0x47b5481dbefa4fa4ull,
  0xcbbb9d5dc1059ed8ull, 0x629a292a367cd507ull,

                         0x152fecd8f70e5939ull,
  0x67332667ffc00b31ull, 0x8eb44a8768581511ull,
  0xdb0c2e0d64f98fa7ull, 0x47b5481dbefa4fa4ull,
  0xcbbb9d5dc1059ed8ull, 0x629a292a367cd507ull,
  0x9159015a3070dd17ull,

  0x67332667ffc00b31ull, 0x8eb44a8768581511ull,
  0xdb0c2e0d64f98fa7ull, 0x47b5481dbefa4fa4ull,
  0xcbbb9d5dc1059ed8ull, 0x629a292a367cd507ull,
  0x9159015a3070dd17ull, 0x152fecd8f70e5939ull };

u128 sandwlvlc512[20] =
{ 0x6a09e667f3bcc908ull, 0xbb67ae8584caa73bull,
  0x3c6ef372fe94f82bull, 0xa54ff53a5f1d36f1ull,
  0x510e527fade682d1ull, 0x9b05688c2b3e6c1full,
  0x1f83d9abfb41bd6bull, 0x5be0cd19137e2179ull,

                         0xbb67ae8584caa73bull,
  0x3c6ef372fe94f82bull, 0xa54ff53a5f1d36f1ull,
  0x510e527fade682d1ull, 0x9b05688c2b3e6c1full,
  0x1f83d9abfb41bd6bull, 0x5be0cd19137e2179ull,
  0x6a09e667f3bcc908ull,

  0x3c6ef372fe94f82bull, 0xa54ff53a5f1d36f1ull,
  0x510e527fade682d1ull, 0x9b05688c2b3e6c1full,
  0x1f83d9abfb41bd6bull, 0x5be0cd19137e2179ull,
  0x6a09e667f3bcc908ull, 0xbb67ae8584caa73bull,
```

```
                        0xa54ff53a5f1d36f1ull,
  0x510e527fade682d1ull, 0x9b05688c2b3e6c1full,
  0x1f83d9abfb41bd6bull, 0x5be0cd19137e2179ull,
  0x6a09e667f3bcc908ull, 0xbb67ae8584caa73bull,
  0x3c6ef372fe94f82bull,

  0x510e527fade682d1ull, 0x9b05688c2b3e6c1full,
  0x1f83d9abfb41bd6bull, 0x5be0cd19137e2179ull,
  0x6a09e667f3bcc908ull, 0xbb67ae8584caa73bull,
  0x3c6ef372fe94f82bull, 0xa54ff53a5f1d36f1ull };


// process a block of message
// may trigger processing of higher levels if necessary
void sandwdoit( sandwstate *hstate )
{ int i; u128 w;
  // check if there's higher level work to do
  if (hstate->lvl1cnt==10)
  { if (hstate->lvl2cnt==100)
    { for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[44+i],hstate->prev[52+i]);
      if (!hstate->lvl3flag)  // setup level 3 starting round constants
      for (i=0;i<20;i++) hstate->prev[60+i] = x128((hstate->lvlc)[i],hstate->prev[i]);
      sandwcmprs(hstate->msg,hstate->prev,3,xtrnds);
      hstate->lvl2cnt=0; hstate->blkn2 = a128(hstate->blkn2,k128(0,1));
      hstate->lvl3flag=1; }
    if (!hstate->lvl2cnt)  // setup level2 superblock starting round constants
    { for (i=0;i<20;i++) hstate->prev[40+i] = x128(c128((hstate->lvlc)[i]),hstate->prev[i]);
      for (i=0;i<20;i+=4) hstate->prev[40+i] = x128(hstate->prev[40+i], hstate->blkn2); }
    for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[24+i],hstate->prev[32+i]);
    sandwcmprs(hstate->msg,hstate->prev,2,xtrnds);
    hstate->lvl1cnt=0; hstate->blkn1 = a128(hstate->blkn1,k128(0,1));
    hstate->lvl2cnt++; }
  for (i=0;i<8;i++) hstate->msg[i] = k128(0,0);
  for (i=0;i<128;i++)   // pack 128 message bytes into eight 128-bit words
  { w = k128(0,hstate->msgbytes[i]);
    hstate->msg[i>>4] = x128(hstate->msg[i>>4], r128(w,120-((i&15)<<3))); }
  hstate->mbcnt = 0;   // reset byte count to 0.
  // if lvl0flag = 0, we're doing level 0
```

```
  if (hstate->lvl0flag==0)
  { sandwcmprs(hstate->msg,hstate->prev,0,xtrnds); hstate->lvl0flag = 1; }
  else
  { if (!hstate->lvl1cnt)  // setup level1 superblock starting round constants
    { for (i=0;i<20;i++) hstate->prev[20+i] = x128((hstate->lvlc)[i],hstate->prev[i]);
      for (i=0;i<20;i+=4) hstate->prev[20+i] = x128(hstate->prev[20+i], hstate->blkn1); }
    sandwcmprs(hstate->msg,hstate->prev,1,xtrnds); hstate->lvl1cnt++; };
 }

// hashsize must be either 384 or 512
void sandwinit( sandwstate *hstate, int hashsize )
{ int i; u128 *rndc;
  //  for (i=0;i<128;i++) hstate->msgbytes[i] = 0;  // not actually needed
  hstate->mbcnt = 0;
  //  for (i=0;i<8;i++) hstate->msg[i] = k128(0,0);  // not actually needed
  hstate->msglen = k128(0,0);
  hstate->blkn1 = k128(0,0); hstate->blkn2 = k128(0,0);
  hstate->lvl0flag = 0;
  hstate->lvl1cnt = 0; hstate->lvl2cnt = 0;
  hstate->lvl3flag = 0;
  hstate->outsize = hashsize;
  if (hashsize==384) hstate->lvlc = sandwlvlc384; else hstate->lvlc = sandwlvlc512;
  // setup level 0 constants
  for (i=0;i<20;i++) hstate->prev[i] = (hstate->lvlc)[i]; }

// bits of data must be a multiple of 8, except for last partial byte
// update is responsible for zeroing unused low order bits in last byte
// mbcnt points to first unused byte, including after any partial byte
void sandwupdate( sandwstate *hstate, int bitsofdata, unsigned char *data )
{ int j;
  for (j=0;j<bitsofdata;)
  { if (hstate->mbcnt>=128) sandwdoit(hstate); // we've got a full message block to process
    hstate->msgbytes[hstate->mbcnt] = *data; data++; j+=8;
    if (j>bitsofdata) // handle partial byte
      hstate->msgbytes[hstate->mbcnt] &= 0xFF<<(j-bitsofdata);
    hstate->mbcnt++; }
  hstate->msglen = a128(hstate->msglen,k128(0,bitsofdata)); }
```

```
// do padding, process any partial block, propagate up levels
void sandwfinish( sandwstate *hstate )
{ int i, padbitpos;
  // process message buffer if full
  if (hstate->mbcnt>=128 && !(hstate->msglen.lo & 7)) sandwdoit(hstate);
  for (i=hstate->mbcnt;i<128;i++) hstate->msgbytes[i]=0; // zero out unused message bytes
  padbitpos = hstate->msglen.lo & 7;
  if (!padbitpos) hstate->mbcnt++;
  hstate->msgbytes[hstate->mbcnt-1] |= 0x80 >> padbitpos;
  sandwdoit(hstate);
  // setup level4 round constants
  for (i=0;i<20;i++) hstate->prev[80+i] = c128((hstate->lvlc)[i]);
  for (i=0;i<20;i+=4) hstate->prev[80+i] = x128(hstate->prev[80+i], hstate->msglen);
  // see if we've only used level 0
  if (hstate->lvl1cnt==0 && hstate->blkn1.hi==0 && hstate->blkn1.lo==0)
   for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[4+i],hstate->prev[12+i]);
  // or if we've only used level 1
  else if (hstate->lvl2cnt==0 && hstate->blkn2.hi==0 && hstate->blkn2.lo==0)
   for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[24+i],hstate->prev[32+i]);
  // or if we've only used level 2
  else if (hstate->lvl3flag==0)
   for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[44+i],hstate->prev[52+i]);
  else  // we've used level 3
   for (i=0;i<8;i++) hstate->msg[i] = x128(hstate->prev[64+i],hstate->prev[72+i]);
  sandwcmprs(hstate->msg,hstate->prev,4,xtrnds);
  // final hash in prev[96-99]
 }
```

Appendix C Test Vectors

SANDstorm-224 Test Vectors

512-bit Test Message:
1111111122222222333333334444444455555555666666667777777788888888

Level 0:

Data:
d0 = 3131313131313131
d1 = 3232323232323232
d2 = 3333333333333333
d3 = 3434343434343434
d4 = 3535353535353535
d5 = 3636363636363636
d6 = 3737373737373737
d7 = 3838383838383838

Message Schedule:
d0 = 3131313131313131
d1 = 3232323232323232
d2 = 3333333333333333
d3 = 3434343434343434
d4 = 0404040404040404
d5 = 0404040404040404
d6 = 0404040404040404
d7 = 0C0C0C0C0C0C0C0C
d8 = 4BDC9850A0353DF8
d9 = 20EA7C9FE27D0FE9
d10 = 18D489624D625048
d11 = D5D3999E28BB32ED
d12 = 690BA6200C78C149
d13 = 15C0EB16A87446A7
d14 = 608C99700F4F587B
d15 = C1844321240D5A9C
d16 = 915E29DFD4F8F6A2
d17 = 5DBA7280B61294D5

d18 = 648A0176ED183441
d19 = 1F80E7F83C787462
d20 = A2B4369645143CC2
d21 = D921A411EFA0DABA
d22 = BC5B533D3B3ACFE7
d23 = D1A0D40D90B73760
d24 = 916BB765DB952B25
d25 = C7665A85E005DB1E
d26 = 33A40D668640540A
d27 = 0A38256244D79D4F
d28 = 446F1E8A3B54F1A4
d29 = 059D86DB28CFD962
d30 = A71581CDEEAF4B9C
d31 = 3D00D34CAD62B7BF

Input Constants:
c0 = C1059ED8367CD507 3070DD17F70E5939 FFC00B3168581511 64F98FA7BEFA4FA4
c1 = 367CD5073070DD17 F70E5939FFC00B31 6858151164F98FA7 BEFA4FA4C1059ED8
c2 = 3070DD17F70E5939 FFC00B3168581511 64F98FA7BEFA4FA4 C1059ED8367CD507
c3 = F70E5939FFC00B31 6858151164F98FA7 BEFA4FA4C1059ED8 367CD5073070DD17
c4 = FFC00B3168581511 64F98FA7BEFA4FA4 C1059ED8367CD507 3070DD17F70E5939

State Outputs:
s1 = E4CF35D9A384C0DE 30940B2061B63E74 27A8E24D8A18AFEC D92399D1CB5A33FB
s2 = 2589B540E0961804 2E85DE9E32669AB9 ED2127C17A18ABD1 00A213587A15CF77
s3 = 1078CD89C3018649 823E9B43A1A77736 59F959B16A104986 FCE2A784B5640F60
s4 = A841F50C071399D0 72880A355B248AE8 10B941345EF2236E 9AE887480CAA0148

Level 1:

Data:

d0 = 0000000000000080
d1 = 0000000000000000
d2 = 0000000000000000
d3 = 0000000000000000
d4 = 0000000000000000
d5 = 0000000000000000
d6 = 0000000000000000
d7 = 0000000000000000

Message Schedule:
d0 = 0000000000000080
d1 = 0000000000000000
d2 = 0000000000000000
d3 = 0000000000000000
d4 = 0000000000000080
d5 = 0000000000000000
d6 = 0000000000000000
d7 = 0000000000000000
d8 = 06A8323C783B359E
d9 = F776DC4EE455E471
d10 = 5737990323C687FB
d11 = BB0F4AEF595DDFE5
d12 = 4843ADADDDB6F0AE
d13 = 64BDAC06062D7C1F
d14 = 8A910A912625980C
d15 = 1FCD1D9FAF9BC591
d16 = 436C5FA217111F65
d17 = D8065E9D789F3852
d18 = F066850BCBCDF9B6
d19 = D8D7A8EED411BDA3
d20 = A8494C384CEE2513

d21 = 8C5761D3908EE848
d22 = 0266B7EEA3977211
d23 = 427A4E1106E23A08
d24 = 799C0425C17C3A6D
d25 = 99A87D0328EDEAC4
d26 = 57D7FD8D28EABC1D
d27 = 0B356E19F4AC4171
d28 = D70A5B5898CA17CC
d29 = 97569A2C98A2DC7E
d30 = 8CC0F7F5C3D11ADF
d31 = 1704D769EF7D6681

Input Constants:
c0 = 69446BD4316F4CD7 42F8D722AC2AD3D1 EF794A0536AA367F FE1108EFB2504EEC
c1 = D2B3E0DE93F41DC9 C79A52199E763545 4FF0F75CEEE1204B 67D9D6750A5FAD23
c2 = 15F968571798413D D145D5AF5A3E8FA8 89D8A866C4E2E475 C1A78D804C691A70
c3 = E77694B03CC18D78 EA668E52C55EF891 E7031615AB15D75E CA9E72838514D277
c4 = 5781FE3D6F4B8CC1 16718592E5DEC54C D1BCDFEC688EF669 AA985A5FFBA45871

State Outputs:
s1 = 78A16FB6195442CA 5F73F45215A31C53 32E71690EEC7DEAC 57D7046BA52A98E7
s2 = 242B21138DA27AF7 0DACFB13504C6D23 E6C3B333F94809F9 578FEE51B6E5E79F
s3 = 7BD3ABE949247649 CA0A36529321E984 5A6A880040C9F970 4A9E619AF6531909
s4 = EFBB50C76A033203 660E19A95E9E663A F9460EBA30C35E80 7BA67B0923A2DE97

Level 4:

Data:
d0 = 0372C45F50703483
d1 = 9579C2008682F5D7
d2 = 688D9E90AE0E27DC

d3 = 1D4965F1537981EE
d4 = CB9071D4E7A148F4
d5 = 6BA2E2BA0ED20B19
d6 = 1F85BD89C98B5779
d7 = 2C29955895473908

Message Schedule:
d0 = 0372C45F50703483
d1 = 9579C2008682F5D7
d2 = 688D9E90AE0E27DC
d3 = 1D4965F1537981EE
d4 = C8E2B58BB7D17C77
d5 = FEDB20BA8850FECE
d6 = 77082319678570A5
d7 = 3160F0A9C63EB8E6
d8 = 9607F1888E0B00BA
d9 = DF092757CB31D737
d10 = 18FF09330717BF62
d11 = 950BB9656921B1CB
d12 = 0B02894DD4BF2FD8
d13 = 544058A94AE23ACC
d14 = C2A233418EC8B559
d15 = D5B0EE6B7A9336AC
d16 = A994C035FEC2B0DE
d17 = AE9C6D1DA14612EE
d18 = 7DCD0962C782E1BC
d19 = B6029CF671EE7592
d20 = 04F5A60071F8998B
d21 = E30542F06F002065
d22 = 6CAD2445F99C2326
d23 = 6DA7EF1930EF608D

d24 = 96BE66791B19014E
d25 = 4551CC1844139E3F
d26 = E74FD132F6C005C1
d27 = 453B8C8AAB6C7E9B
d28 = 14575C796415626A
d29 = C8710B7165730910
d30 = 433177016272D525
d31 = 452B0C329BA07BFB

Input Constants:
c0 = 3EFA6127C98328F8 CF8F22E808F1A4C6 003FF4CE97A7E8EE 9B0670584105B25B
c1 = C9832AF8CF8F20E8 08F1A6C6003FF6CE 97A7EAEE9B067258 4105B05B3EFA6327
c2 = CF8F22E808F1A4C6 003FF4CE97A7E8EE 9B0670584105B25B 3EFA6127C98328F8
c3 = 08F1A6C6003FF6CE 97A7EAEE9B067258 4105B05B3EFA6327 C9832AF8CF8F20E8
c4 = 003FF4CE97A7E8EE 9B0670584105B25B 3EFA6127C98328F8 CF8F22E808F1A4C6

State Outputs:
s1 = 53D409A03A58F75F 82375626B5C4C31C B50F55BE38C08874 820CF2FE10F9E6C8
s2 = B733F3781BEC47D2 EE1C6214F4CF79A5 1960C27EFA7EFB5A 6F02344941E19A1C
s3 = 8D2BF541DFCC4EB9 34229367EE2B42FE 88887A39C8494F99 40029CC9FAC9BBB9
s4 = 3863B976947B4264 69ACCA283851F4C4 5FC7F0DC25DD602F 69CD6855193563B9

SANDstorm-256 Test Vectors

512-bit Test Message:
1111111122222222333333334444444455555555666666667777777788888888

Level 0:

Data:

d0 = 3131313131313131
d1 = 3232323232323232
d2 = 3333333333333333
d3 = 3434343434343434
d4 = 3535353535353535
d5 = 3636363636363636
d6 = 3737373737373737
d7 = 3838383838383838

Message Schedule:
d0 = 3131313131313131
d1 = 3232323232323232
d2 = 3333333333333333
d3 = 3434343434343434
d4 = 0404040404040404
d5 = 0404040404040404
d6 = 0404040404040404
d7 = 0C0C0C0C0C0C0C0C
d8 = 4BDC9850A0353DF8
d9 = 20EA7C9FE27D0FE9
d10 = 18D489624D625048
d11 = D5D3999E28BB32ED
d12 = 690BA6200C78C149
d13 = 15C0EB16A87446A7
d14 = 608C99700F4F587B
d15 = C1844321240D5A9C
d16 = 915E29DFD4F8F6A2
d17 = 5DBA7280B61294D5
d18 = 648A0176ED183441
d19 = 1F80E7F83C787462
d20 = A2B4369645143CC2

d21 = D921A411EFA0DABA
d22 = BC5B533D3B3ACFE7
d23 = D1A0D40D90B73760
d24 = 916BB765DB952B25
d25 = C7665A85E005DB1E
d26 = 33A40D668640540A
d27 = 0A38256244D79D4F
d28 = 446F1E8A3B54F1A4
d29 = 059D86DB28CFD962
d30 = A71581CDEEAF4B9C
d31 = 3D00D34CAD62B7BF

Input Constants:
c0 = 6A09E667BB67AE85 3C6EF372A54FF53A 510E527F9B05688C 1F83D9AB5BE0CD19
c1 = BB67AE853C6EF372 A54FF53A510E527F 9B05688C1F83D9AB 5BE0CD196A09E667
c2 = 3C6EF372A54FF53A 510E527F9B05688C 1F83D9AB5BE0CD19 6A09E667BB67AE85
c3 = A54FF53A510E527F 9B05688C1F83D9AB 5BE0CD196A09E667 BB67AE853C6EF372
c4 = 510E527F9B05688C 1F83D9AB5BE0CD19 6A09E667BB67AE85 3C6EF372A54FF53A

State Outputs:
s1 = B22F978A3BCC10EE 7FDD30C32C63C56E 626C25283356C4E9 FFBF10622106381C
s2 = 00906C3BD392F370 C07FAA5124D35370 D2F79AABDFFFEF97 45DAD8DB5D64383E
s3 = DFBC9A38851FD03C 8A20A18B2724A280 B9797CA20D54C958 12A7838729688E11
s4 = 2EF815C396C20A84 462F43E653D6834B FADD9D6B6081CE91 5E7A71EE5688AD9A

Level 1:

Data:
d0 = 0000000000000080
d1 = 0000000000000000
d2 = 0000000000000000

d3 = 0000000000000000
d4 = 0000000000000000
d5 = 0000000000000000
d6 = 0000000000000000
d7 = 0000000000000000

Message Schedule:
d0 = 0000000000000080
d1 = 0000000000000000
d2 = 0000000000000000
d3 = 0000000000000000
d4 = 0000000000000080
d5 = 0000000000000000
d6 = 0000000000000000
d7 = 0000000000000000
d8 = 06A8323C783B359E
d9 = F776DC4EE455E471
d10 = 5737990323C687FB
d11 = BB0F4AEF595DDFE5
d12 = 4843ADADDDB6F0AE
d13 = 64BDAC06062D7C1F
d14 = 8A910A912625980C
d15 = 1FCD1D9FAF9BC591
d16 = 436C5FA217111F65
d17 = D8065E9D789F3852
d18 = F066850BCBCDF9B6
d19 = D8D7A8EED411BDA3
d20 = A8494C384CEE2513
d21 = 8C5761D3908EE848
d22 = 0266B7EEA3977211
d23 = 427A4E1106E23A08

d24 = 799C0425C17C3A6D
d25 = 99A87D0328EDEAC4
d26 = 57D7FD8D28EABC1D
d27 = 0B356E19F4AC4171
d28 = D70A5B5898CA17CC
d29 = 97569A2C98A2DC7E
d30 = 8CC0F7F5C3D11ADF
d31 = 1704D769EF7D6681

Input Constants:
c0 = 44F1F3A42DA5A401 7A41B094F6997671 ABD3CF14FB84A61D 41F9A8450D686083
c1 = 0948390F07A2E39C DA92C5F97D6D9711 F9694DA42CD51D42 A45FDD7B4B0FDE7B
c2 = 3CFE9F4976DD064A 9171F82EBFD63BFC CD744300841F228E 2FD33EBCE60396BB
c3 = 7AF36F02D4118243 1125C90738A77B2B E299B1BB675D2F3F A9C02D0215067D63
c4 = 7FF647BC0DC76208 59AC9A4D08364E52 90D47B0CDBE66014 6214829CF3C758A0

State Outputs:
s1 = A26B38082775FD0E B699B80C00F2F296 381311CDFD2967D8 E164241A0B7A078A
s2 = CA366E1322197604 7A8CA74D6B233766 F17186994230B966 34EF4F5E54D27E9C
s3 = ECF30909F973BF57 61FA5D21B062F63F AE7AE448B842A2A8 C7771329ADBA85D2
s4 = 5211203F5B39A3AD 4B19406A9E44DF08 9CCB663BC1B73F96 403C8630824DD4FB

Level 4:

Data:
d0 = 4E983101DE064259
d1 = D763E52DB09004A9
d2 = 9669F585456BC570
d3 = 26133733A6C08258
d4 = 98274E2C7920D5A9
d5 = 3195E727F567E86E

d6 = 6DBAE0A2838786F0
d7 = 74D3C96ED69FAA67

Message Schedule:
d0 = 4E983101DE064259
d1 = D763E52DB09004A9
d2 = 9669F585456BC570
d3 = 26133733A6C08258
d4 = D6BF7F2DA72697F0
d5 = E6F6020A45F7ECC7
d6 = FBD31527C6EC4380
d7 = 52C0FE5D705F283F
d8 = 25D7F004BBC5068D
d9 = 12298626ED464E99
d10 = 928E146219FF08C0
d11 = FB7A8DBFE0C54DF2
d12 = 68A3587019416B20
d13 = B0AF510290D05293
d14 = FFE7DC5412B0F4E3
d15 = D585CD25E08432A8
d16 = 644C0CCFCAA1098D
d17 = 4A55F7E99E225FAD
d18 = 8947BE6AC1D15C12
d19 = DA642E9545AD7E28
d20 = F9BC734435D2EF05
d21 = 1C99AA4163959524
d22 = 3AEED6DC86A62643
d23 = AB807F7A62DE2268
d24 = 39C1B808106E5818
d25 = 0ACFF08B7528D334
d26 = 4A13327D877B7C44

d27 = F3E6BBD6AF6EA035
d28 = 8D0CE9C36223A9AB
d29 = 0DFEAF26248DC5AA
d30 = E8F8065B431B0124
d31 = 881FABC305789E8A

Input Constants:
c0 = 95F619984498537A C3910C8D5AB008C5 AEF1AD8064FA9573 E07C2654A41F30E6
c1 = 4498517AC3910E8D 5AB00AC5AEF1AF80 64FA9773E07C2454 A41F32E695F61B98
c2 = C3910C8D5AB008C5 AEF1AD8064FA9573 E07C2654A41F30E6 95F619984498537A
c3 = 5AB00AC5AEF1AF80 64FA9773E07C2454 A41F32E695F61B98 4498517AC3910E8D
c4 = AEF1AD8064FA9573 E07C2654A41F30E6 95F619984498537A C3910C8D5AB008C5

State Outputs:
s1 = 7F4F77AB7EF2B3DF 679CF5C7E9E23E35 C365ADAC6F64CDC6 DC33438251CDC54E
s2 = 2C7AEA05DAF433E6 DBE6E0FBD43505A2 E2025DF96602DDDC F9FA6F92D127E90D
s3 = 0B7425987B3AFE52 D889193E260C515A D9B570101739DB9C 134BD17BC3C7BAB0
s4 = D11D3EC6372732C3 0CA684DA52A467FA 04FD67AE59CC0551 027F7FB21BB36FE8