ESSENCE: A Candidate Hashing Algorithm for the NIST Competition

Jason Worth Martin

January 12, 2009

Abstract

This paper gives the technical specification for the ESSENCE cryptographic hashing algorithm submitted to the NIST competition for selecting SHA-3. ESSENCE is a hybrid design using Merkle hash trees combined with Merkle-Damgård iterative hashing structures. The size of each component to be hashed with the Merkle-Damgård construction and the height of the trees are parameterized to allow for a selection which balances parallel versus serial performance in specific applications. The ESSENCE compression functions can be implemented in a constant time form which is immune to cache-timing based attacks or in a faster form using look-up tables. Both forms feature extensive instruction-level parallelism to take advantage of SIMD instructions available on modern processors.

The additional implementation submitted to the NIST competition using thread-based parallelism together with hand tuned assembly code was capable of hashing messages of greater than eight megabytes at 12.1 cycles/byte on a quad-core, Xeon-based Linux server.

1 Overview

Figure 1 provides an overview of the ESSENCE hashing construction. ESSENCE is a hybrid design in which the data to be hashed is sub-divided into a small collection of large, equal sized, components. Each of these components, which we refer to as a Merkle-Damgård Block (abbreviated MD Block), is hashed separately using a Merkle-Damgård based iterative construction with varying initialization vector. The resulting sub-hashes are combined with Merkle hashing tree structures, and the resulting root hashes of each tree are combined again with a final Merkle-Damgård structure called the running hash. The running hash is padded with a final block containing the length of the data hashed and the algorithm parameters used. The size of each Merkle-Damgård Block and the height of the trees are parameterized to allow for a selection which balances parallel versus serial performance in specific applications. Section 4 gives a list of all the algorithm parameters, their meanings, and their default values.

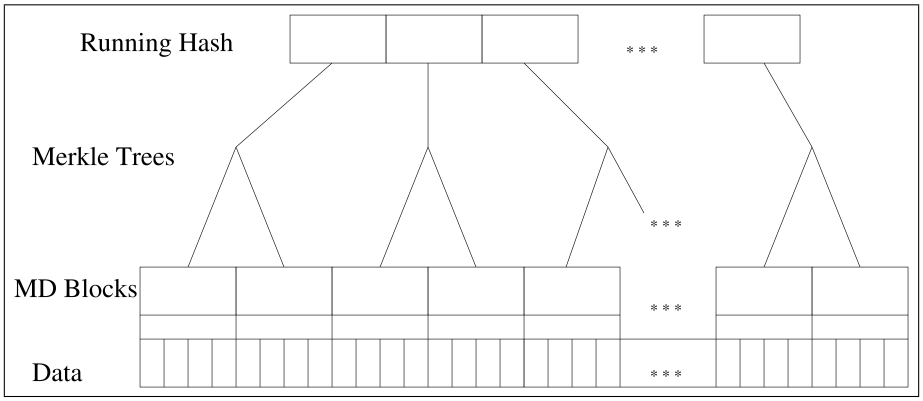

Figure 1: Overview of ESSENCE Construction

The compression functions used for ESSENCE are detailed in the accompanying document *ESSENCE: A Family of Cryptographic Hashing Algorithms* where the full mathematical description and analysis of the compression functions are given. There are two compression functions: one operates on 256-bit blocks and the other on 512-bit blocks. Both compression functions are Davies-Meyer constructions based on key-dependent permutations E_{256} and E_{512}. The 256-bit and 512-bit functions differ only in the size of the registers used (32-bit registers for the 256-bit function and 64-bit registers for the 512-bit) and L, the linear combining function.

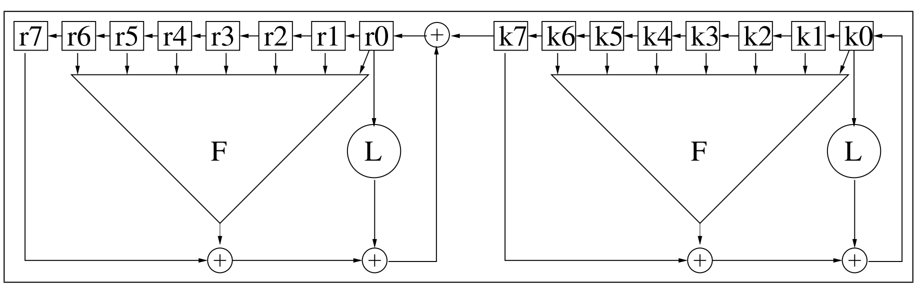

Figure 2: The ESSENCE Compression Function Logic

Figure 2 provides an overview of the compression function stepping logic. Rather than using look-up tables for the non-linear portion of the logic, the non-linear function F is computed with each step and takes constant time. The linear function, L, can be implemented in constant time via a linear feedback shift register in Galois configuration, or it can be accelerated using a look-up table.

The trade off is that the accelerated version may be vulnerable to cache-timing attacks. Since cryptographic hashing algorithms are often used to hash secret data (such as when generating cryptographic keys or verifying authentication credentials), implementers should consider resistance to side-channel attacks as well as performance. The details for both approaches are given in Appendix B of *ESSENCE: A Family of Cryptographic Hashing Algorithms*.

ESSENCE internally generates only 256-bit or 512-bit hash sizes but uses truncation combined with different initial values to support varying hash sizes. This is similar to the approach used in the SHA-2 family of hashing algorithms.

1.1 Advantages

Here we list some of the important advantages of ESSENCE.

1. **High Server-Platform Performance**

   The ESSENCE design has been optimized for parallel implementations. Therefore, it runs best on large platforms featuring both high instruction-level parallelism (such as the vector processing SSE instructions in the x86_64 family or the very-long-instruction-word paradigm of the Itanium family) and high thread-level parallelism (such as multi-cored platforms). On a quad-core Xeon-based machine (a typical server platform), ESSENCE can hash at 12.1 clock cycles per byte of data.

2. **Good Constant-Time Implementations Possible**

   ESSENCE can easily be implemented to run in constant time. This thwarts timing attacks such as those demonstrated by Bernstein against AES implementations in [Ber04]. On a quad-core Xeon-based machine, a constant-time implementation of ESSENCE can hash at 46.2 clock cycles per byte of data.

3. **Very Scalable**

   ESSENCE is also extremely scalable, so as future platforms featuring more parallelism become available, it will show even better performance characteristics.

4. **Very Simple Embedded Implementation**

   The ESSENCE design was also deliberately constrained to ensure that it can be reasonably implemented on 8-bit embedded platforms; the compression function can be implemented entirely from AND, XOR, NOT, and SHIFT. Furthermore, if look-up tables are used to implement the linear function, then there is no need to shift bits across byte boundaries. This allows for very simple 8-bit implementations.

5. **Simple Hardware Implementation of Compression Functions**

   The only non-linear function is F whose prime implicants are listed in Appendix A of *ESSENCE: A Family of Cryptographic Hashing Algorithms*. The shift-register based design then makes the compression function extremely simple to implement in hardware.

6. **Designed to Defend Against Linear and Differential Cryptanalysis**

   As described in sections 3.5 and 3.6 of *ESSENCE: A Family of Cryptographic Hashing Algorithms*, ESSENCE has been specifically designed to be resistant to differential and linear cryptanalysis.

7. **Well Established Design Principles**

   The compression functions are based entirely on the theory of shift register sequences. The non-linear function, F, is used to drive non-linear feedback shift registers, and the linear function, L, which combines the non-linear streams is implemented as a linear feedback shift register. The theory of shift registers sequences is old and well developed.

   The compression functions are then used in Merkle-Damgård based iterative chaining structures and Merkle hash trees. Both structures are thoroughly established and well studied.

   The Merkle-Damgård structure also allows ESSENCE to be used in other Merkle-Damgård based protocols such as HMAC, pseudo-random number generation, etc.

8. **Tunable Security Parameter**

   ESSENCE provides a security parameter in the number of steps used within the compression function. The minimum number of steps is 24 (cryptanalysis becomes possible if fewer than 24 steps are used and trivial if fewer than 16 steps are used). The recommended number is 32, which is the value used for the NIST competition. However, implementations may choose to use a larger number (which must be a multiple of 8). The run time of the algorithm scales linearly with the number of steps used.

9. **Tunable Parallelism Parameter**

   The size of the Merkle hash trees used in ESSENCE is tune-able. Since the NIST competition appears to focus primarily on sequential performance, the value used for the NIST competition is 0 (corresponding to only one Merkle-Damgård block per tree, so each tree consists of only a root node). However, applications needing a greater level of parallelism (e.g. whole file-system hashing, distributed peer-to-peer based systems, etc.) may choose much larger tree structures (and store intermediate results) to give far superior performance and prevent re-hashing entire data sets when only a small portion has been modified.

1.2 Disadvantages

Here we list what we consider to be the most serious disadvantages of ESSENCE.

1. **Requires Assembly and Parallel Programming for High Speed Implementations**

   Because the C programming language does not allow the programmer to explicitly describe parallelism, it is not possible to get good performance from pure C language implementations of ESSENCE. Assembly programming is required to take advantage of the instruction-level parallelism and to access vector instructions. Parallel programming constructs such as OpenMP, MPI, or Threads are required to take advantage of the thread-level parallelism designed into ESSENCE. In short, fast ESSENCE implementations are highly non-trivial. An example of such an implementation using hand tuned assembly language and OpenMP is provided in the Additional Implementations submitted to the NIST competition.

2. **Slower Performance on Short Messages**

   The very designed structure which allows ESSENCE to take advantage of parallelism adds overhead. For short messages, the overhead involved in padding and other bookkeeping can be significant. Hence ESSENCE performs poorly when processing very short messages.

3. **Slower Performance on 8-bit Processors**

   Even though ESSENCE was designed to be simple to implement on resource-limited processors, the implementation will necessarily be much slower than on a large register processor. ESSENCE was designed to take full advantage of instruction-level parallelism by using larger register sizes. On small processors (with only 8 or 16-bit registers), the processor will have to execute many more instructions to compensate for the smaller register size. We estimate an execution rate of 2124 cycles per byte of message size on an 8-bit processor.

4. **Common Merkle-Damgård Based Weaknesses**

   Since ESSENCE is based on Merkle-Damgård iterative chaining, it shares the weaknesses inherent in Merkle-Damgård based hashing algorithms.

5. **No Mathematical Proof of Reduction to a Known Hard Problem**

   Some current proposed hashing algorithms provide mathematical proofs of security by demonstrating a reduction to a known hard problem (e.g. finding a minimal length vector in a lattice, discrete logarithms, factorization, etc.). Unfortunately, ESSENCE does not lend itself to such a reduction. Although the compression function is shown to be resistant to differential and linear cryptanalysis in *ESSENCE: A Family of Cryptographic Hashing Algorithms*, this does not prove that it is secure.

2 Endian Conventions

Since ESSENCE must treat bytes of data as 32 or 64-bit integers, it is important to have a convention for the byte ordering. We have chosen to use Little Endian byte ordering. In other words, the least significant byte of an integer will be the byte located in the smallest memory address. This is a pragmatic choice based in the prevalence of the x86 architecture which uses Little Endian storage for multi-byte integers. Figure 3 illustrates the difference between Big and Little Endian conventions by showing the way the same sequence of eight bytes in memory would be interpreted as a 64-bit integer (qword), two 32-bit integers (dword), or four 16-bit integers (word).

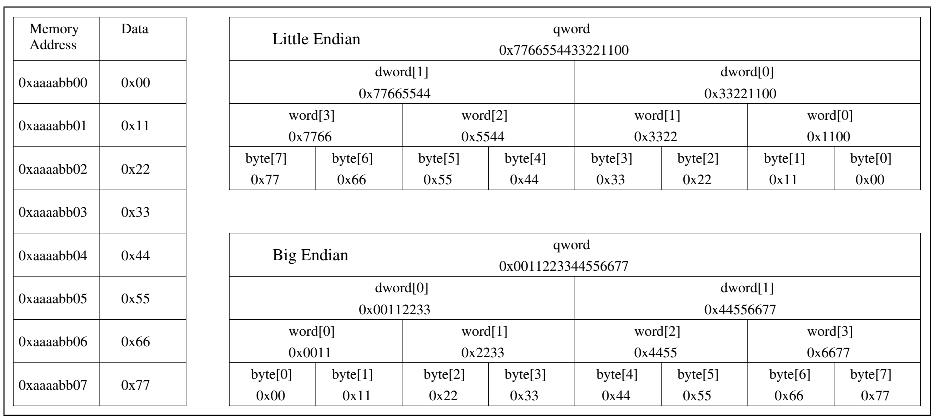

Figure 3: Byte Ordering

Although ESSENCE uses a Little Endian convention, that does not prevent it from running on Big Endian processors, it simply requires a thoughtful implementation. (The reference implementation submitted to the NIST competition runs correctly on Big and Little Endian processors. The additional implementation submitted has a configuration parameter which is used to indicate the Endianness of the target platform.)

3 Some Overloaded Definitions

Within this document we overload the word “block” with many different meanings. We will attempt to conform to the following definitions.

Block: We use the term “block” to indicate either 256 or 512 bits of data. If we are discussing a hash size of greater than 256 bits, then a “block” refers to 512 bits of data. Otherwise, it refers to 256 bits of data.

Merkle-Damgård Block: A "Merkle-Damgård block" is, in general, composed of many "blocks". We will occasionally abbreviate "Merkle-Damgård block" as "MD block". A large message is split into many "Merkle-Damgård blocks". Each "Merkle-Damgård block", except, perhaps, the last one, is of a fixed size given by the ESSENCE algorithm parameter ESSENCE_MD_BLOCK_SIZE_IN_BYTES (see section 4.2).

Complete Merkle-Damgård Block: We call a Merkle-Damgård block "complete" if its size is exactly ESSENCE_MD_BLOCK_SIZE_IN_BYTES bytes.

Last Merkle-Damgård Block: The "last Merkle-Damgård block" consists of any remaining data which is not contained in the previous complete Merkle-Damgård blocks. The last block of the "last Merkle-Damgård block" is padded with zeros, if needed, to reach the block boundary. In other words, the padding will be less than 256 bits if the block size is 256 bits and less than 512 bits if the block size is 512 bits.

Final Block: The term "final block" does not refer to the last block of message data. The term "final block" is reserved for a specially formated block of information which encodes the algorithm parameters and data length. The "final block" is not appended to the data, and it is not hashed together with the data within a Merkle-Damgård block. Instead, the "final block" is hashed at the end of the "running hash". (See section 7 for complete details.) In particular, the "final block" prevents length extension attacks as demonstrated in the example in section 7.1.

4 Algorithm Parameters

In this section we describe the algorithm parameters. It is important to note that the value of the hash is dependent upon the algorithm parameters used.

4.1 Number of Steps in Compression Function

Name: ESSENCE_COMPRESS_NUM_STEPS

Default Value: 32

ESSENCE_COMPRESS_NUM_STEPS is the security parameter of the algorithm. It defines the number of iterations of the update logic in the ESSENCE compression function. As shown in *ESSENCE: A Family of Cryptographic Hashing Algorithms*, a value of less than 24 is insecure, and the value used should be a multiple of 8. For the NIST competition the value of this constant will be 32.

4.2 Size of Merkle-Damgård Block

Name: ESSENCE_MD_BLOCK_SIZE_IN_BYTES

Default Value: 1048576

This is the size, in bytes, to use for the Merkle-Damgård Blocks. The MD Block sizes must be multiples of 64 to ensure that the resulting blocks have a data size in bits that is divisible by 512. For the NIST competition the value of this constant will be 1048576 which is equal to 2^{20}.

4.3 Size of Merkle Hash Trees

Name: ESSENCE_HASH_TREE_LEVEL

Default Value: 0

The ESSENCE_HASH_TREE_LEVEL defines the "level" or "height" of the Merkle hash trees used in the given implementation. There is a trade-off between serial and parallel performance. Larger hash trees allow for greater parallelism at the cost of a slower serial implementation. Likewise, smaller hash trees result in faster serial implementations, but less parallelism. The ESSENCE_HASH_TREE_LEVEL changes the resulting value of the hash, so it must be a fixed standard agreed upon for the given use. The number of Merkle-Damgård blocks hashed within a given tree is $2^{\text{ESSENCE\_HASH\_TREE\_LEVEL}}$. So, a tree height of zero means there is only one MD block per tree. This parameter is restricted to values between 0 and 255 inclusive. For the NIST competition the value of this constant will be 0.

4.4 Small Organizational Constant

Name: ESSENCE_ORGANIZATIONAL_SMALL_CONSTANT

Default Value: 0xb7e15162

The point of this parameter (and the following one) is give an organization the flexibility to produce implementations of the algorithm that are unique to the organization, project, network, server etc. The choices of values for these to constants is arbitrary. There are no known "weak" choices. NOTE: These values can not be considered secret, proprietary, or otherwise protected since they can be easily recovered.

ESSENCE_ORGANIZATIONAL_SMALL_CONSTANT is an unsigned 32-bit integer. For the NIST competition, the value for this constant shall be the first 8 hexadecimal digits of the fractional part of the base-16 expansion of the Euler constant e.

4.5 Big Organizational Constant

Name: ESSENCE_ORGANIZATIONAL_BIG_CONSTANT

Default Value: 0x8aed2a6abf715880

As with the previous parameter, this parameter provides flexibility for implementations.

ESSENCE_ORGANIZATIONAL_BIG_CONSTANT is an unsigned 64-bit integer. For the NIST competition, the value for this constant shall be the next 16 hexadecimal digits of the fractional part of the base-16 expansion of e.

5 Merkle-Damgård Blocks

Let us clarify some terminology. When we refer to a Merkle-Damgård block, we mean a very large block of data which will be hashed using a Merkle-Damgård construction. When we refer only to a "block" we mean a small amount of data, either 256 or 512 bits, which is used as input into the compression function in the Merkle-Damgård construction. Hence, a "Merkle-Damgård block" is divided into "blocks" on which the compression function operates.

The data to be hashed is divided into Merkle-Damgård blocks where each one, except perhaps the last one, is ESSENCE_MD_BLOCK_SIZE_IN_BYTES bytes long. Each Merkle-Damgård block is divided into blocks of either 256 or 512 bits depending upon the size of the requested hash. Hashes 256-bits or less will use the 256-bit compression function with 256-bit blocks. Hashes of greater than 256-bits will use the 512-bit compression function with 512-bit blocks. The blocks are then hashed using a Merkle-Damgård construction with an initialization vector that depends upon the block number and algorithm parameters. Let

$$G(R, K) = G(R, K, \text{ESSENCE\_COMPRESS\_NUM\_STEPS})$$

denote the compression function as described in section 4 of *ESSENCE: A Family of Cryptographic Hashing Algorithms*, then the construction is given by:

$$\begin{aligned} H_0 &= \mathrm{IV}_b \\ H_i &= G(H_{i-1}, M_{i-1}) \end{aligned}$$

where IV_b is the initialization vector for Merkle-Damgård block number b (see below), and M_i is a 256 or 512-bit block. The final H_i is the value of the hash of the Merkle-Damgård block. Note that each M_i is defined to be an array of eight integers (of either 32 or 64-bits), whereas the data is a stream of bytes. We use Little Endian byte ordering to interpret the bytes as integers.

Except for the last Merkle-Damgård block, there is no padding used on the Merkle-Damgård blocks. This is because each Merkle-Damgård block is of the same size (except perhaps the last). The total number of Merkle-Damgård blocks, the size of each Merkle-Damgård block, and the length of the last block are incorporated into the "final block" of the running hash (see section 7) to prevent length extension attacks.

The last Merkle-Damgård block is simply the remaining data. If needed, the last block of the last Merkle-Damgård block is padded with zeros to ensure that it is the correct size for the compression function. Length extension attacks are prevented by including the data bit length in the "final block" of the running hash.

5.1 Merkle-Damgård Block Initialization Vector

The first 32 bytes of the Merkle-Damgård block initialization vector is the same for the 256-bit and 512-bit compression function. Here are the values for the first 32 bytes.

Bytes 0-7: An unsigned 64-bit integer representing the Merkle-Damgård block number in Little Endian format.

Bytes 8-15: An unsigned 64-bit integer representing the number of bytes used in each Merkle-Damgård block. This integer is in Little Endian format. This is the ESSENCE_MD_BLOCK_SIZE_IN_BYTES parameter.

Byte 16-17: An unsigned 16-bit integer representing the size, in bits, of the hash requested. This integer is in Little Endian format. This value is taken from the hashbitlen value.

Byte 18: An unsigned 8-bit integer representing the number of steps the update logic in the compression function will use. This is the value of the parameter ESSENCE_COMPRESS_NUM_STEPS.

Byte 19: An unsigned 8-bit integer representing the level of the Merkle hash trees to be used. This is the ESSENCE_HASH_TREE_LEVEL parameter.

Bytes 20-23: An unsigned 32-bit integer. This integer is in Little Endian format. This value is taken from the organizational algorithm parameter ESSENCE_ORGANIZATIONAL_SMALL_CONSTANT.

Bytes 24-31: An unsigned 64-bit integer. This integer is in Little Endian format. This value is taken from the organizational algorithm parameter ESSENCE_ORGANIZATIONAL_BIG_CONSTANT.

If the 512-bit compression function is being used, then the next 32 bytes are initialized from the hexadecimal expansion of the fractional part of pi. They begin with the 64^{th} digit and are as follows:

Bytes 32-39: The unsigned 64-bit integer constant 0x452821e638d01377 in Little Endian.

Bytes 40-47: The unsigned 64-bit integer constant 0xbe5466cf34e90c6c in Little Endian.

Bytes 48-55: The unsigned 64-bit integer constant 0xc0ac29b7c97c50dd in Little Endian.

Bytes 56-63: The unsigned 64-bit integer constant 0x3f84d5b5b5470917 in Little Endian.

6 Merkle Hash Trees

For the purposes of the NIST competition, this section may be ignored since the Merkle hash trees are effectively not used (each tree consists only of a root node whose value is just the hash of the associated Merkle-Damgård block).

The Merkle hash trees are included in the definition of ESSENCE because they allow for a very high degree of parallelism in the hashing structure, and ESSENCE provides a parameter, ESSENCE_HASH_TREE_LEVEL, which determines the size of the trees to use. Larger hash trees favor greater parallelism at the expense of slower sequential implementations. Since the NIST competition is primarily based on sequential performance, we choose to set this parameter to zero for the competition. The remainder of this sections discusses how ESSENCE is defined to operate for non-trivial Merkle hash trees. We assume that the reader is familiar with binary trees and the related terminology, data structures, and algorithms.

For our purposes, a Merkle hash tree is a binary tree whose leaf nodes contain the values of the Merkle-Damgård block hashes (see Figure 4). The value of each node is then defined in terms of the value of its children (if it has any) according to the following rules:

1. A node exists if and only if at least one of its children exists or it is a leaf.
2. The value of a leaf node is the hash of the corresponding Merkle-Damgård block.
3. The value of a non-leaf node is:
   (a) The JOIN of the values of its children if both children exist.
   (b) The value of its child if only one child exists.

We define the JOIN of values of two nodes as:

$$\mathrm{JOIN}(X, Y) = G(G(\mathrm{IV}, X), Y)$$

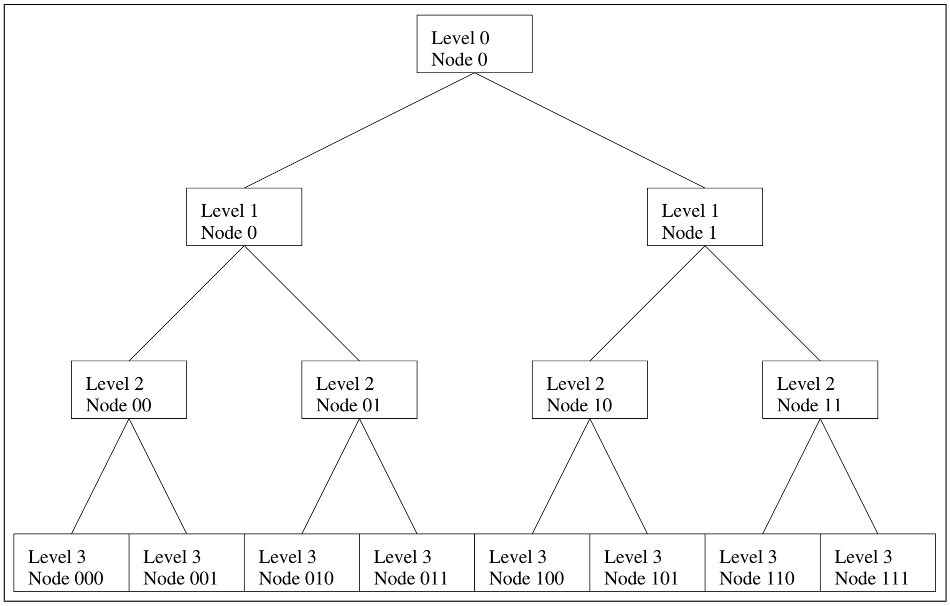

Figure 4: A Level 3 Tree

where G is the compression function as defined above and IV is an initialization vector. Table 1 gives the initialization vector, IV, used for the JOIN for the 256-bit and 512-bit case. The IV values are taken from the hexadecimal expansion of the fractional part of π. Note that the 256-bit values have been chosen so that in Little Endian representation they correspond to the first 4 integers in the 512-bit IV.

The rules allow for incomplete trees to have a well defined value.

IV	512-bit case	256-bit case
IV[0]	0x243f6a8885a308d3	0x85a308d3
IV[1]	0x13198a2e03707344	0x243f6a88
IV[2]	0xa4093822299f31d0	0x03707344
IV[3]	0x082efa98ec4e6c89	0x13198a2e
IV[4]	0x452821e638d01377	0x299f31d0
IV[5]	0xbe5466cf34e90c6c	0xa4093822
IV[6]	0xc0ac29b7c97c50dd	0xec4e6c89
IV[7]	0x3f84d5b5b5470917	0x082efa98

Table 1: JOIN Initialization Vector Values

7 Running Hash

The resulting values of the root nodes of all the Merkle hash trees are combined sequentially in a final Merkle-Damgård construction we call the "running hash". Since there are two possible block sizes, either 256-bit or 512-bit, the running hash has an initialization vector (IV) which may take one of two different possible values depending upon which block size is used. The IV used is the same as the IV for the JOIN operation described in the previous section. Table 1 gives the IV values, which are just the hexadecimal expansion of the fractional part of π.

After all data has been hashed, the running hash will compress one last block which we refer to as the "final block". The final block contains some of the algorithm parameters, the number of complete Merkle-Damgård blocks, and the length in bits of the final incomplete Merkle-Damgård block (which may be zero). Together, this encodes the total data length. The final block prevents length extension attacks (see section 7.1 below). The final block looks similar to the Merkle-Damgård block initialization vectors, but with the number of complete Merkle-Damgård blocks and data length of last block replacing some values. The precise description of the final block is below.

The final block is different depending upon if it is 256 or 512 bits. In either case, the first 32 bytes are the same:

Bytes 0-7: An unsigned 64-bit integer representing the number of complete Merkle-Damgård blocks processed. If ESSENCE is requested to hash a message of zero length, then all bits in this field will be set to ones.

Bytes 8-15: An unsigned 64-bit integer representing the number of bytes used in each Merkle-Damgård block. This integer is in Little Endian format. This is the ESSENCE_MD_BLOCK_SIZE_IN_BYTES parameter.

Byte 16-17: An unsigned 16-bit integer representing the size, in bits, of the hash requested. This integer is in Little Endian format. This value is taken from the hashbitlen value.

Byte 18: An unsigned 8-bit integer representing the number of steps the update logic in the compression function will use. This is the value of the parameter ESSENCE_COMPRESS_NUM_STEPS.

Byte 19: An unsigned 8-bit integer representing the level of the Merkle hash trees to be used. This is the ESSENCE_HASH_TREE_LEVEL parameter.

Bytes 20-23: An unsigned 32-bit integer. This integer is in Little Endian format. This value is take from the organizational algorithm parameter ESSENCE_ORGANIZATIONAL_SMALL_CONSTANT.

Bytes 24-31: An unsigned 64-bit integer representing the number of bits hashed in the final incomplete Merkle-Damgård block. If the size of the total data hashed is divisible by the Merkle-Damgård block size, then there is no incomplete Merkle-Damgård block, so the value of this integer would be zero.

If the 512-bit compression function is being used, then the next 32 bytes are initialized from the hexadecimal expansion of the fractional part of pi. They begin with the 64^{th} digit and are as follows:

Bytes 32-39: The unsigned 64-bit integer constant 0x452821e638d01377 in Little Endian.

Bytes 40-47: The unsigned 64-bit integer constant 0xbe5466cf34e90c6c in Little Endian.

Bytes 48-55: The unsigned 64-bit integer constant 0xc0ac29b7c97c50dd in Little Endian.

Bytes 56-63: The unsigned 64-bit integer constant 0x3f84d5b5b5470917 in Little Endian.

7.1 Resistance to Length Extension Attacks

A length extension attack against a standard Merkle-Damgård construction works as follows:

A user computes $Z = \text{hash}(X)$ for some secret message X, and makes public the resulting hash output Z. An attacker may then use that hash output to compute $\text{hash}(X||Y)$ for some Y.

Even if the hash algorithm uses Merkle-Damgård strengthening, the attacker may set

$$Y = \{\text{any padding bits used for X}\}||\{\text{any other value}\}$$

and compute $\text{hash}(X||Y)$ by starting from the hash value Z, and computing the hash of any additional bits from there, complete with new padding and Merkle-Damgård strengthening which reflects the length of $X||Y$.

What prevents this attack on ESSENCE is that the "final block" is not simply appended to the message. Hashing in ESSENCE (using the default parameters for the trees, etc., given in section 4) occurs at two layers. The lowest level is hashed in Merkle-Damgård fashion, but then the results of those hashes are used within a separate "running hash" at the top level.

Let us illustrate by continuing the example:

Let X be the secret message. Suppose for simplicity that it is a single 512-bit block (let's assume we're requesting a 512-bit hash from ESSENCE). There won't be any padding.

Let $G(a, b)$ be the ESSENCE compression function with a playing the role of the chaining variable, and b the next input block.

Let D be the initial vector for the Merkle-Damgård layer of ESSENCE, and let R be the initial vector for the running hash layer, and let FB denote the final block. Then,

$$Z = \text{essence}(X) = G(G(R, G(D, X)), FB).$$

Note the order of the compression function calls.

Now, let's say that an attacker wishes to find essence$(X||Y)$ where Y is another 512-bit block. In ESSENCE, that looks like:

$$\text{essence}(X||Y) = G(G(R, G(G(D, X), Y)), FB2)$$

where $FB2$ is the final block for this new message. Look very closely at the order! No matter what choice is made for Y, Z does not appear in this expression. For example, we might be thinking of just letting Y be equal to the first final block, FB. In that case, we have:

$$\text{essence}(X||Y) = \text{essence}(X||FB) = G(G(R, G(G(D, X), FB)), FB2).$$

Let's ignore the last compression for the moment, and just look at the next-to-last one, it is

$$G(R, G(G(D, X), FB)).$$

Comparing that to

$$Z = G(G(R, G(D, X)), FB)$$

shows that the order of the compression function calls has changed. So, $Z =$ essence(X), never appears as the value of the running hash chaining variable when computing essence$(X||Y)$, thus the simple length extension attack cannot be applied to ESSENCE.

8 Expected Strength

Since ESSENCE is primarily a Merkle-Damgård based design, its strength against first pre-image and first collision attacks is based in the compression functions. In *ESSENCE: A Family of Cryptographic Hashing Algorithms* we show that for both the 256-bit and 512-bit compression functions, a minimum of 24 steps is required to provide resistance to differential and linear cryptanalysis. We expect that the compression function will be vulnerable to those attacks with fewer than 24 steps (and trivially vulnerable with fewer than 16 steps). As a matter of caution, we recommend 32 steps. The number of steps taken must be a multiple of eight.

With at least 32 steps used for the compression functions, we expect that the amount of work required to find a first collision will be on the order of a brute force search ($2^{n/2}$ for an n-bit hash). Likewise we expect that the work required to find a first pre-image will be on the order of a brute force search (2^n for an n-bit hash).

9 Computational Efficiency

9.1 Memory Required

If the accelerated version of the L function is required, then the implementation must include a table of 256 4-byte entries (for L_{32}) and a table of 256 8-byte entries (for L_{64}). This requires a minimum of 3072 bytes for the tables, but the actual size used in the reference platform was seen to be 3448 bytes (including symbol tables and padding for data alignment).

The executable code size is vastly dependent upon compilation parameters and which libraries are linked in. The reference implementation required approximately 24,000 bytes on a platform similar to the NIST reference platform.

The amount of RAM required for variable storage during execution (including stack space) is also implementation dependent. We estimate that with the reference implementation fewer than 16,000 bytes are required. More advanced implementation using parallel methods may require substantially more RAM and dynamic memory allocation support.

9.2 Measured Performance on Intel Core 2 Platforms

Appendix A gives the results of timing tests on three Intel Core 2 Platforms. The platforms tested were Windows Vista 32-bit and Windows Vista 64-bit on a duel core Intel Core 2 machine similar to the NIST reference platform and a 64-bit Linux distribution on a quad core Xeon based machine. The specifications for the test platforms are also given in Appendix A. We tested the serial ANSI C based implementation as well as a parallel version using C with OpenMP and x86_64 assembly code. Table 2 summarize the results for message sizes of over eight megabytes. Note that we measure performance in processor clock cycles per byte of message size which is independent of the clock frequency.

Platform	Implementation	Hash Size	cycles/byte
Vista 32 Core 2 Dual Core	Serial C-only	224	150.8
		256	149.8
		384	176.5
		512	176.5
Vista 64 Core 2 Dual Core	Serial C-only	224	63.7
		256	63.6
		384	64.2
		512	64.2
	OpenMP and Assembly	224	19.7
		256	19.5
		384	23.5
		512	23.5
Linux 64 Xeon Quad Core	OpenMP and Assembly	224	10.3
		256	9.9
		384	12.1
		512	12.1
	OpenMP and Assembly Constant Time	224	22.4
		256	22.1
		384	46.2
		512	46.2

Table 2: Summary of Performance on Core2 Based Platforms

9.3 Estimated Performance on 8-bit Platforms

We did not have access to an 8-bit development platform on which to test ESSENCE. However, based on the performance of the serial C code on the 32-bit Intel Core 2 platform we estimate

$$\text{8-bit performance} \geq 177 \cdot 4 \cdot 3 = 2124 \text{ cycles/byte}.$$

(The multiple of three is based on the assumption that an 8-bit processor can only execute one instruction per clock cycle while the Core 2 can average three instructions per cycle.)

9.4 Hardware Estimates

At the time of this writing, no hardware implementations have been constructed. However, we can give some very gross bounds in the upper and lower limits of the number of gates needed to implement the stepping logic in the compression function. One difficulty, though, is the definition of a “logic gate” for the purposes of comparing estimates. Without having a fixed technology for the hardware implementation we do not know if “logic gate” indicates only a single, 2-input NAND gate, or if "logic gate" might refer to something as complicated as a 38-input XOR gate. In the following sections we provide estimates for both, but we caution the reader that these are very broad estimates.

9.4.1 Conservative Estimate

We give here estimates of the gate count for the stepping logic to implement a single step of the compression function. For the purposes of making conservative estimates, we will assume that multi-input logic gates are constructed from 2-input logic gates, and we will give our gate count estimates in terms of 2-input logic gates. (e.g. We assume that a 7-input AND gate will be constructed from six 2-input AND gates. While we know this is not strictly true, it gives an upper bound on the number of gates required.)

The only non-linear function is F whose prime implicants are listed in Appendix A of *ESSENCE: A Family of Cryptographic Hashing Algorithms*. The F function requires 63 prime implicants, each with seven factors. Hence, the prime implicants require $63 \cdot 6 = 378$ 2-input AND gates. The 63 prime implicants can then be combined with 62 2-input OR gates. So, each bit of output requires at most 440 2-input logic gates.

So, for the 256-bit compression function, F requires $32 \cdot 440 = 14080$ gates. For the 512-bit compression function, F requires $64 \cdot 440 = 28160$ gates.

Each bit of output from the linear function L_{64} is dependent on at most 38 bits of input. A 38-input XOR gate can be constructed from 37 2-input XOR gates. So L_{64} uses at most $64 \cdot 37 = 2368$ gates.

Each bit of output from the linear function L_{32} is dependent on at most 19 bits of input. A 19-input XOR gate can be constructed from 18 2-input XOR gates. So L_{32} uses at most $32 \cdot 18 = 576$ gates.

Table 3 summarizes the result.

	F function	L function	Total
256-bit	14,080	576	14,656
512-bit	28,160	2,368	30,528

Table 3: Conservative Estimates Using Only 2-Input Logic Gates

9.4.2 Optimistic Estimates

The conservative estimates given in the previous section assume that all logic is being implemented with 2-input gates. If, on the other hand, we assume that we have logic gates of whatever size we need, then the estimates become much nicer.

In this case, each bit of output of the F function requires 63 7-input AND gates, and a single 63-input OR gate, for a total of 64 gates per bit. So, the 256-bit compression function requires 2048 gates, and the 512-bit compression function requires 4096 gates.

The linear function L_{32} requires 32 19-input XOR gates.

The linear function L_{64} requires 64 38-input XOR gates.

The results are summarized in Table 4.

	F function	L function	Total
256-bit	2,048	32	2,080
512-bit	4,096	64	4,160

Table 4: Optimistic Gate Count Estimates

10 Reference Implementation

The "Reference Implementation" is purely expository and is intended for debugging and testing purposes. The reference implementation has been written in ANSI C. However, the current ANSI C standard, C99, is fully supported by very few compilers. So, it is worth mentioning that the only C99 feature required by the reference implementation is support of the `long long` data type for 64-bit integers. In the ANSI C99 standard, the data types `int32_t`, `uint32_t`, `int64_t`, and `uint64_t` are defined in "stdint.h". We would prefer to simply used the data types defined in "stdint.h" since this ensures the greatest portability. However, we discovered that some widely used development platforms did not have the "stdint.h" file available even though a `long long` 64-bit integer data type was supported. To make the reference implementation available on the widest possible range of compilers, we define `int32_t`, `uint32_t`, `int64_t`, and `uint64_t` in `essence_api.h` based on the assumption that `int` is a 32-bit integer and `long long` is a 64-bit integer. If these assumptions are incorrect for a particular target platform, then the `typedef` statements in `essence_api.h` should be modified. If the target platform supports the C99 `stdint.h` data types, then `essence_api.h` should be modified to include the `stdint.h` header file and the superfluous definitions can be removed. Table 5 summarizes the data size assumptions

The file `essence_api.h` also defines a constant called ESSENCE_DEBUG_LEVEL which controls the amount of debugging output generated by the implementation.

`typedef` in `essence_api.h`	Defined as	Assumed to be
`DataLength`	`unsigned long long`	64-bit unsigned integer
`uint64_t`	`unsigned long long`	64-bit unsigned integer
`int64_t`	`long long`	64-bit signed integer
`uint32_t`	`unsigned int`	32-bit unsigned integer
`int32_t`	`int`	32-bit signed integer

Table 5: Data Size Assumptions

11 Additional Implementation

The "Additional Implementation" includes x86_64 assembly language and OpenMP parallel C code and is intended to demonstrate how to take advantage of the instruction-level and thread-level parallelism present in ESSENCE. Also included in the additional implementation are versions of the compression functions which use the constant-time calculation of the linear function, L, instead of look-up tables. The additional implementation has been designed with a modular approach allowing for compilation on a wide range of platforms (including non-x86_64 platforms and Big Endian platforms). The compilation options are controlled by constants declared in `essence_api.h` and compiler flags. Subsections 11.1 through 11.5 describes the constants and their effects.

The x86_64 assembly code included has been written for use with the YASM assembler [Joh08]. YASM was chosen because it allowed for a single assembly file to be used to generate object files for use with Windows Vista, Mac OS X, and Linux. YASM is freely available open source software. The NIST submission for ESSENCE includes source code and binary executables for YASM. We wish to acknowledge Peter Johnson and Brian Gladman for their assistance with YASM.

11.1 Debug Level

Name: ESSENCE_DEBUG_LEVEL

This controls the level of debugging code that is compiled in. Level zero prevents any debug code from being compiled, and does not require that "stdio.h" and the corresponding C libraries be available. Each bit of ESSENCE_DEBUG_LEVEL controls a different type of debugging output. The bits are given in Table 6 in Little Endian order (i.e. bit 0 is the least significant bit).

ESSENCE_DEBUG_LEVEL bit	Description
0	The hash “state” variable is printed at the end of every call to Init, Update, or Final.
1	The hash state is printed at the end of every call to Merge_Tree_256 or Merge_Tree_512.
2	The hash state is printed at the end of every call to Join_256 or Join_512.
3	The values of all intermediate computations are printed within the compression functions. NOTE: this level of output is only available with basic C language versions of the compression functions. It is not supported in the constant time versions or in the assembly language versions. Also note that this will generate a tremendous amount of output!

Table 6: ESSENCE_DEBUG_LEVEL Bit Usage

11.2 Use Constant Time Code

Name: ESSENCE_USE_CONSTANT_TIME_CODE

If ESSENCE_USE_CONSTANT_TIME_CODE is set to 1, then the code will link against the constant-time versions of the compression functions. The constant-time versions do not use look-up tables and are immune to cache-based timing attacks. This option makes the code somewhat slower (it takes approximately three to four times as long to hash the same amount of data) however it is much more secure and should be used for any implementation in which secret data is being hashed.

11.3 Use Parallel Code

Name: ESSENCE_USE_PARALLEL_CODE

If ESSENCE_USE_PARALLEL_CODE is set to 1, then we assume that the code is being compiled using OpenMP for a target with large memory and CPU resources. This will enable options that optimize for speed on large, highly-parallel platforms. However, these options are much slower on resource-limited platforms. Note that the compiler probably needs to receive flags to build with OpenMP. See the included Makefiles for details.

Also note: Microsoft Visual Studio requires `vcomp90.dll` for the resulting OpenMP based executable to run. This `dll` is not part of the standard Vista distribution, but it is included in the Microsoft Visual C++ Redistributable Package which is available free of charge from Microsoft. However, the terms of the redistribution license are too restrictive to allow this `dll` to be included in the NIST submission for ESSENCE. Therefore, anyone wishing to compile the OpenMP enabled version of ESSENCE with Microsoft Visual Studio must obtain `vcomp90.dll` independently and place it in the same directory (folder) where the ESSENCE executable is located.

11.4 Assume Little Endian

Name: ESSENCE_ASSUME_LITTLE_ENDIAN

If this option is set to 1, then we assume that we are on a Little Endian platform and optimize accordingly.

11.5 Use Core2 Assembly Code

Name: ESSENCE_USE_CORE2_ASSEMBLY

If this option is set to 1, then we assume that we are using the Intel Core 2 assembly code. This option requires that the assembly code be assembled and linked against. Note that YASM, our assembler, must have the correct options passed in to tell it what type of platform to target. See the included Makefiles for details.

12 Intellectual Property Statements

I, Jason Worth Martin, am not aware of any patents applicable to the ESSENCE hashing algorithm. I do not intend to pursue any patents on ESSENCE, and I wish for it to be available royalty free, world-wide, with no restrictions.

13 Full Disclosure

I, Jason Worth Martin, worked for the Naval Research Laboratory from 1996 through 1999 in a COMSEC group. As a result of my work with classified cryptographic algorithms, I am required to submit any potentially sensitive papers to the National Security Agency's pre-publication review process. I submitted *ESSENCE: A Family of Cryptographic Hashing Algorithms* for pre-publication review, and it was cleared for public release. At no time did the NSA or any other government organization request that I make any modifications to the algorithm or the paper. The design of ESSENCE has not been influenced in any way by the NSA.

References

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming`, 2004.

[Fog08] Agner Fog. *Software Optimization Manuals*, 2008. `http://www.agner.org/optimize`.

[Joh08] Peter Johnson. *The Yasm Modular Assembly Project*, 2008. `http://www.tortall.net/projects/yasm`.

A Timing Data

Included in the "Additional Implementation" section of the NIST submission for ESSENCE are x86_64 assembly language implementations for the compression functions (tuned to the Intel Core 2 micro-architecture) and parallel implementations, using OpenMP, of the Update function. The assembly code has been written to work with 64-bit Windows, Linux, and Mac OS X by using the Yasm assembler ([Joh08]). The additional implementation has header file configuration parameters which, together with compiler flags and build configuration, can be used to select various levels of optimization for the resulting code. In this section we give the timing results for some of these configurations.

The program "speed_test.c" which is included in the "Additional Implementation" was used to generate these timing data. The timing is for the "Hash" function which performs all-at-once hashing and includes the time for "Init" and "Final". Therefore, we feel that the timing data is a very accurate reflection of the total performance of ESSENCE in real world systems.

The timing analysis in these sections was performed by using the "read time stamp counter" or "rdtsc" instruction available on the x86 platform. This machine instruction returns the value of the "time stamp counter" which is a 64-bit unsigned counter that is incremented once per processor clock cycle. By reading the counter once before performing the hash and once after, the total number of processor clock cycles used during the hash can be computed. This is the preferred method for timing execution on a x86 platform. We observed that other timing methods (such as using the ANSI C standard library functions included from the "time.h" header file) were often grossly inaccurate. An excellent description of software timing on the x86 architecture is given by Agner Fog in [Fog08].

We report our timing information in "clock cycles" and "clock cycles per byte" because we believe that is the most meaningful description of an algorithm's efficiency. Processors with the same architecture but clocked at different rates will still have similar "cycles/byte" performance.

For brevity in the remaining sections, Table 7 lists the test platforms together with a short name we will use to describe them in the following sections.

Vista32 (Vista Desktop)

- **Processor:** Intel Core2 Duo E8400 (Dual Core)
- **Clock Freq.:** 3.00 GHz
- **RAM:** 4 GB
- **OS:** Windows Vista Ultimate 32-bit
- **C Compiler:** Visual Studio Professional Edition 2008

Vista64 (Vista Server)

- **Processor:** Intel Core2 Duo E8400 (Dual Core)
- **Clock Freq.:** 3.00 GHz
- **RAM:** 4 GB
- **OS:** Windows Vista Ultimate 64-bit
- **C Compiler:** Visual Studio Professional Edition 2008 (using 64-bit target)
- **Assembler:** Yasm version 0.7.1.2093 for Windows

Linux64 (Linux Server)

- **Processor:** Intel Xeon E5320 (Quad Core)
- **Clock Freq.:** 1.86 GHz
- **RAM:** 16 GB
- **OS:** Ubuntu (Hardy Heron 64-bit Server)
- **C Compiler:** gcc version 4.2.3
- **Assembler:** Yasm version 0.7.1.2093 for Linux

Table 7: Test Platforms

A.1 Serial C-only Code

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	52542.00	52542.000
2	24543.00	12271.500
4	23967.00	5991.750
8	23940.00	2992.500
16	23859.00	1491.188
32	23994.00	749.813
64	30816.00	481.500
128	45252.00	353.531
256	73908.00	288.703
512	131004.00	255.867
1024	245232.00	239.484
2048	473274.00	231.091
4096	937521.00	228.887
8192	1849536.00	225.773
16384	3757473.00	229.338
32768	7380783.00	225.244
65536	14852241.00	226.627
131072	29416581.00	224.431
262144	59096646.00	225.436
524288	118173843.00	225.399
1048576	173950551.00	165.892
2097152	314679636.00	150.051
4194304	631810395.00	150.635
8388608	1258976709.00	150.082
16777216	2530043235.00	150.802
33554432	5062205646.00	150.865
67108864	10614809142.00	158.173
134217728	20442744342.00	152.310
268435456	40323408885.00	150.216
536870912	80645706153.00	150.214

Table 8: Vista32 Serial C-only Code: 224 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	21528.00	21528.000
2	16245.00	8122.500
4	16191.00	4047.750
8	15921.00	1990.125
16	15930.00	995.625
32	15786.00	493.313
64	20574.00	321.469
128	30195.00	235.898
256	49086.00	191.742
512	87327.00	170.561
1024	163251.00	159.425
2048	315522.00	154.063
4096	620352.00	151.453
8192	1229346.00	150.067
16384	2447856.00	149.405
32768	4884993.00	149.078
65536	9759699.00	148.921
131072	19653237.00	149.942
262144	39230316.00	149.652
524288	78464556.00	149.659
1048576	156936114.00	149.666
2097152	314040825.00	149.746
4194304	627730083.00	149.663
8388608	1254825810.00	149.587
16777216	2513877678.00	149.839
33554432	5027582601.00	149.834
67108864	10055769912.00	149.843
134217728	20120653458.00	149.911
268435456	40771364418.00	151.885
536870912	80434574253.00	149.821

Table 9: Vista32 Serial C-only Code: 256 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	101583.00	101583.000
2	36243.00	18121.500
4	35883.00	8970.750
8	35757.00	4469.625
16	35676.00	2229.750
32	35730.00	1116.563
64	36693.00	573.328
128	47016.00	367.313
256	69534.00	271.617
512	114588.00	223.805
1024	204687.00	199.890
2048	384894.00	187.937
4096	745614.00	182.035
8192	1466487.00	179.015
16384	2908467.00	177.519
32768	5792571.00	176.775
65536	11708415.00	178.656
131072	23178447.00	176.838
262144	46336833.00	176.761
524288	92482965.00	176.397
1048576	185049117.00	176.477
2097152	369443169.00	176.164
4194304	739405584.00	176.288
8388608	1481332599.00	176.589
16777216	2964869199.00	176.720
33554432	5920507953.00	176.445
67108864	11847876471.00	176.547
134217728	23677354197.00	176.410
268435456	47716990668.00	177.760
536870912	95044054518.00	177.033

Table 10: Vista32 Serial C-only Code: 384 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	45810.00	45810.000
2	36891.00	18445.500
4	35964.00	8991.000
8	35793.00	4474.125
16	35793.00	2237.063
32	35910.00	1122.188
64	36855.00	575.859
128	47169.00	368.508
256	69678.00	272.180
512	114741.00	224.104
1024	204822.00	200.021
2048	385056.00	188.016
4096	746208.00	182.180
8192	1467054.00	179.084
16384	2909439.00	177.578
32768	5794272.00	176.827
65536	11717190.00	178.790
131072	23185116.00	176.888
262144	47559780.00	181.426
524288	92537460.00	176.501
1048576	185820732.00	177.212
2097152	369701865.00	176.288
4194304	740017260.00	176.434
8388608	1478803347.00	176.287
16777216	2969442207.00	176.993
33554432	5924039652.00	176.550
67108864	11846315466.00	176.524
134217728	23686544520.00	176.479
268435456	47366473680.00	176.454
536870912	95320476252.00	177.548

Table 11: Vista32 Serial C-only Code: 512 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	40896.00	40896.000
2	11934.00	5967.000
4	11349.00	2837.250
8	11115.00	1389.375
16	11214.00	700.875
32	11592.00	362.250
64	14067.00	219.797
128	20232.00	158.063
256	32355.00	126.387
512	56601.00	110.549
1024	105183.00	102.718
2048	202338.00	98.798
4096	403470.00	98.503
8192	791226.00	96.585
16384	1572399.00	95.972
32768	3138147.00	95.769
65536	6220620.00	94.919
131072	12683286.00	96.766
262144	25601670.00	97.663
524288	50051709.00	95.466
1048576	100535859.00	95.878
2097152	200337714.00	95.528
4194304	315884169.00	75.313
8388608	533940570.00	63.651
16777216	1067966937.00	63.656
33554432	2140471071.00	63.791
67108864	4281994476.00	63.807
134217728	8606533941.00	64.124
268435456	17163917919.00	63.941
536870912	34409397222.00	64.092

Table 12: Vista64 Serial C-only Code: 224 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	13662.00	13662.000
2	7794.00	3897.000
4	7677.00	1919.250
8	7524.00	940.500
16	7488.00	468.000
32	7866.00	245.813
64	9387.00	146.672
128	13554.00	105.891
256	21591.00	84.340
512	37800.00	73.828
1024	70083.00	68.440
2048	134865.00	65.852
4096	265716.00	64.872
8192	526824.00	64.310
16384	1040760.00	63.523
32768	2353671.00	71.828
65536	4169943.00	63.628
131072	8400546.00	64.091
262144	16596027.00	63.309
524288	33469101.00	63.837
1048576	66836430.00	63.740
2097152	133322382.00	63.573
4194304	266431086.00	63.522
8388608	533262735.00	63.570
16777216	1093745862.00	65.192
33554432	2148291846.00	64.024
67108864	4526712351.00	67.453
134217728	8814521493.00	65.673
268435456	17087096790.00	63.654
536870912	34163343981.00	63.634

Table 13: Vista64 Serial C-only Code: 256 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	70695.00	70695.000
2	14418.00	7209.000
4	14040.00	3510.000
8	13905.00	1738.125
16	13905.00	869.063
32	13878.00	433.688
64	14670.00	229.219
128	18054.00	141.047
256	25974.00	101.461
512	42129.00	82.283
1024	74529.00	72.782
2048	139356.00	68.045
4096	269676.00	65.839
8192	584181.00	71.311
16384	1047321.00	63.923
32768	2084742.00	63.621
65536	4162230.00	63.511
131072	8313399.00	63.426
262144	16920054.00	64.545
524288	33615603.00	64.117
1048576	66850767.00	63.754
2097152	134338158.00	64.057
4194304	269765856.00	64.317
8388608	540336816.00	64.413
16777216	1078780932.00	64.300
33554432	2157495300.00	64.298
67108864	4309556715.00	64.217
134217728	8593515063.00	64.027
268435456	17246226159.00	64.247
536870912	34485393087.00	64.234

Table 14: Vista64 Serial C-only Code: 384 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	19620.00	19620.000
2	14526.00	7263.000
4	13959.00	3489.750
8	13932.00	1741.500
16	13950.00	871.875
32	13941.00	435.656
64	14535.00	227.109
128	18063.00	141.117
256	26118.00	102.023
512	42309.00	82.635
1024	74781.00	73.028
2048	139590.00	68.159
4096	269703.00	65.845
8192	528876.00	64.560
16384	1047906.00	63.959
32768	2086551.00	63.676
65536	4162644.00	63.517
131072	8314146.00	63.432
262144	16691274.00	63.672
524288	34556103.00	65.911
1048576	67321890.00	64.203
2097152	135612738.00	64.665
4194304	270102834.00	64.398
8388608	540844002.00	64.474
16777216	1073242620.00	63.970
33554432	2158172415.00	64.319
67108864	4315229118.00	64.302
134217728	8625762279.00	64.267
268435456	17217749502.00	64.141
536870912	34997906628.00	65.189

Table 15: Vista64 Serial C-only Code: 512 Bit Hash Length

A.2 Parallel C with Assembly Code

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	29160.00	29160.000
2	8703.00	4351.500
4	8370.00	2092.500
8	8325.00	1040.625
16	8307.00	519.188
32	8631.00	269.719
64	10368.00	162.000
128	14850.00	116.016
256	23895.00	93.340
512	42048.00	82.125
1024	78300.00	76.465
2048	150372.00	73.424
4096	300483.00	73.360
8192	589896.00	72.009
16384	1171323.00	71.492
32768	2409138.00	73.521
65536	4675824.00	71.347
131072	9444024.00	72.052
262144	19520856.00	74.466
524288	37757070.00	72.016
1048576	76409658.00	72.870
2097152	92568402.00	44.140
4194304	93318552.00	22.249
8388608	185877378.00	22.158
16777216	332907885.00	19.843
33554432	660749913.00	19.692
67108864	1585242576.00	23.622
134217728	2976319539.00	22.175
268435456	5297908599.00	19.736
536870912	10600642809.00	19.745

Table 16: Vista64 Parallel C with Assembly Code: 224 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	15525.00	15525.000
2	7650.00	3825.000
4	7542.00	1885.500
8	7506.00	938.250
16	7434.00	464.625
32	7731.00	241.594
64	9270.00	144.844
128	13320.00	104.063
256	21303.00	83.215
512	37368.00	72.984
1024	69354.00	67.729
2048	133173.00	65.026
4096	262908.00	64.187
8192	520794.00	63.573
16384	1035441.00	63.198
32768	2164266.00	66.048
65536	4155606.00	63.410
131072	8303553.00	63.351
262144	16466796.00	62.816
524288	32884389.00	62.722
1048576	66897927.00	63.799
2097152	81571095.00	38.896
4194304	81811107.00	19.505
8388608	163024200.00	19.434
16777216	326044881.00	19.434
33554432	650667528.00	19.391
67108864	1308736350.00	19.502
134217728	2607859422.00	19.430
268435456	5222560113.00	19.456
536870912	10438217325.00	19.443

Table 17: Vista64 Parallel C with Assembly Code: 256 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	20187.00	20187.000
2	10737.00	5368.500
4	10395.00	2598.750
8	10422.00	1302.750
16	10431.00	651.938
32	10386.00	324.563
64	10899.00	170.297
128	13311.00	103.992
256	19332.00	75.516
512	31140.00	60.820
1024	54963.00	53.675
2048	102546.00	50.071
4096	197901.00	48.316
8192	388242.00	47.393
16384	768825.00	46.925
32768	1574028.00	48.036
65536	3052593.00	46.579
131072	6096753.00	46.515
262144	12210876.00	46.581
524288	24445044.00	46.625
1048576	48906612.00	46.641
2097152	49161681.00	23.442
4194304	97927560.00	23.348
8388608	196342038.00	23.406
16777216	390873168.00	23.298
33554432	783326313.00	23.345
67108864	1573185105.00	23.442
134217728	3136766508.00	23.371
268435456	6286136454.00	23.418
536870912	12551085612.00	23.378

Table 18: Vista64 Parallel C with Assembly Code: 384 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	16029.00	16029.000
2	10899.00	5449.500
4	10557.00	2639.250
8	10440.00	1305.000
16	10503.00	656.438
32	10458.00	326.813
64	10962.00	171.281
128	13419.00	104.836
256	19350.00	75.586
512	31275.00	61.084
1024	55008.00	53.719
2048	102627.00	50.111
4096	197973.00	48.333
8192	388431.00	47.416
16384	769266.00	46.952
32768	1546758.00	47.203
65536	3054501.00	46.608
131072	6160626.00	47.002
262144	12214053.00	46.593
524288	24492879.00	46.716
1048576	48757005.00	46.498
2097152	48850425.00	23.294
4194304	97885071.00	23.338
8388608	195828786.00	23.345
16777216	392298390.00	23.383
33554432	781805088.00	23.300
67108864	1579094595.00	23.530
134217728	3137515398.00	23.376
268435456	6275520387.00	23.378
536870912	12566398068.00	23.407

Table 19: Vista64 Parallel C with Assembly Code: 512 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	20965.00	20965.000
2	9177.00	4588.500
4	8904.00	2226.000
8	8855.00	1106.875
16	8771.00	548.188
32	9429.00	294.656
64	11256.00	175.875
128	16464.00	128.625
256	26845.00	104.863
512	47642.00	93.051
1024	89208.00	87.117
2048	172319.00	84.140
4096	360304.00	87.965
8192	677061.00	82.649
16384	1348256.00	82.291
32768	2687167.00	82.006
65536	5368251.00	81.913
131072	10731063.00	81.872
262144	21450065.00	81.826
524288	42903560.00	81.832
1048576	86003253.00	82.019
2097152	83972945.00	40.041
4194304	85288231.00	20.334
8388608	156642185.00	18.673
16777216	171827628.00	10.242
33554432	343946687.00	10.250
67108864	687325310.00	10.242
134217728	1373095311.00	10.230
268435456	2745482292.00	10.228
536870912	5489440446.00	10.225

Table 20: Linux64 Parallel C with Assembly Code: 224 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	12257.00	12257.000
2	9660.00	4830.000
4	9247.00	2311.750
8	9065.00	1133.125
16	8841.00	552.562
32	9212.00	287.875
64	11284.00	176.312
128	16492.00	128.844
256	26873.00	104.973
512	47684.00	93.133
1024	89229.00	87.138
2048	172340.00	84.150
4096	339241.00	82.823
8192	671692.00	81.994
16384	1338211.00	81.678
32768	2670598.00	81.500
65536	5333860.00	81.388
131072	10679354.00	81.477
262144	21320152.00	81.330
524288	42633521.00	81.317
1048576	85404347.00	81.448
2097152	82840996.00	39.502
4194304	82902918.00	19.766
8388608	84586159.00	10.083
16777216	166764122.00	9.940
33554432	331725639.00	9.886
67108864	663965428.00	9.894
134217728	1327106249.00	9.888
268435456	2654319528.00	9.888
536870912	5309436762.00	9.890

Table 21: Linux64 Parallel C with Assembly Code: 256 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	38976.00	38976.000
2	10864.00	5432.000
4	10339.00	2584.750
8	10325.00	1290.625
16	10423.00	651.438
32	10409.00	325.281
64	11592.00	181.125
128	13300.00	103.906
256	19208.00	75.031
512	31164.00	60.867
1024	55132.00	53.840
2048	104538.00	51.044
4096	215565.00	52.628
8192	439229.00	53.617
16384	793100.00	48.407
32768	1578360.00	48.168
65536	3155684.00	48.152
131072	6340593.00	48.375
262144	12624199.00	48.157
524288	25247698.00	48.156
1048576	50598534.00	48.255
2097152	50730876.00	24.190
4194304	50806462.00	12.113
8388608	101274271.00	12.073
16777216	202395060.00	12.064
33554432	406531076.00	12.116
67108864	809916884.00	12.069
134217728	1619010379.00	12.063
268435456	3238881667.00	12.066
536870912	6478319701.00	12.067

Table 22: Linux64 Parallel C with Assembly Code: 384 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	16359.00	16359.000
2	12978.00	6489.000
4	12320.00	3080.000
8	12208.00	1526.000
16	12250.00	765.625
32	12341.00	385.656
64	12873.00	201.141
128	15659.00	122.336
256	22561.00	88.129
512	31283.00	61.100
1024	55286.00	53.990
2048	103187.00	50.384
4096	199367.00	48.674
8192	397096.00	48.474
16384	789054.00	48.160
32768	1572669.00	47.994
65536	3157091.00	48.173
131072	6312124.00	48.158
262144	12681410.00	48.376
524288	25233474.00	48.129
1048576	50653701.00	48.307
2097152	50721272.00	24.186
4194304	50764903.00	12.103
8388608	101349206.00	12.082
16777216	202431915.00	12.066
33554432	404832862.00	12.065
67108864	827072561.00	12.324
134217728	1619935905.00	12.069
268435456	3239543118.00	12.068
536870912	6478436328.00	12.067

Table 23: Linux64 Parallel C with Assembly Code: 512 Bit Hash Length

A.3 Parallel C, Assembly Code, Constant Time

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	39536.00	39536.000
2	25186.00	12593.000
4	24857.00	6214.250
8	24766.00	3095.750
16	24647.00	1540.438
32	25634.00	801.062
64	32704.00	511.000
128	49035.00	383.086
256	81298.00	317.570
512	145362.00	283.910
1024	274190.00	267.764
2048	532063.00	259.796
4096	1052457.00	256.948
8192	2082808.00	254.249
16384	4166897.00	254.327
32768	8291206.00	253.028
65536	16573606.00	252.893
131072	33097218.00	252.512
262144	66163986.00	252.396
524288	132330856.00	252.401
1048576	264898354.00	252.627
2097152	185563749.00	88.484
4194304	186950925.00	44.573
8388608	297913833.00	35.514
16777216	390942013.00	23.302
33554432	757087303.00	22.563
67108864	1528187094.00	22.772
134217728	3003617820.00	22.379
268435456	6007080345.00	22.378
536870912	12009593533.00	22.370

Table 24: Linux64 Parallel C, Assembly Code, Constant Time: 224 Bit Hash

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	27643.00	27643.000
2	25802.00	12901.000
4	24773.00	6193.250
8	24773.00	3096.625
16	24738.00	1546.125
32	25410.00	794.062
64	32970.00	515.156
128	48643.00	380.023
256	80836.00	315.766
512	145299.00	283.787
1024	274218.00	267.791
2048	532287.00	259.906
4096	1068963.00	260.977
8192	2085489.00	254.576
16384	4141613.00	252.784
32768	8267434.00	252.302
65536	16524725.00	252.147
131072	33036108.00	252.046
262144	66092698.00	252.124
524288	132089832.00	251.941
1048576	264241803.00	252.001
2097152	185151757.00	88.287
4194304	185406725.00	44.204
8388608	185412850.00	22.103
16777216	371188258.00	22.125
33554432	741925905.00	22.111
67108864	1485137276.00	22.130
134217728	2967730619.00	22.111
268435456	5932775667.00	22.101
536870912	11866576617.00	22.103

Table 25: Linux64 Parallel C with Constant Time Assembly Code: 256 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	56105.00	56105.000
2	36855.00	18427.500
4	36302.00	9075.500
8	36302.00	4537.750
16	36337.00	2271.062
32	36442.00	1138.812
64	37282.00	582.531
128	48370.00	377.891
256	71764.00	280.328
512	118825.00	232.080
1024	213486.00	208.482
2048	402927.00	196.742
4096	781746.00	190.856
8192	1539279.00	187.900
16384	3050173.00	186.168
32768	6075475.00	185.409
65536	12154632.00	185.465
131072	24266998.00	185.143
262144	48428100.00	184.739
524288	96871579.00	184.768
1048576	193736340.00	184.761
2097152	193842894.00	92.431
4194304	195360256.00	46.578
8388608	387746177.00	46.223
16777216	776190205.00	46.265
33554432	1551369057.00	46.234
67108864	3104060701.00	46.254
134217728	6204598008.00	46.228
268435456	12410108935.00	46.231
536870912	24819790569.00	46.230

Table 26: Linux64 Parallel C with Constant Time Assembly Code: 384 Bit Hash Length

Message Size (bytes)	CPU Clock Cycles	Cycles/Byte
1	40348.00	40348.000
2	36883.00	18441.500
4	36456.00	9114.000
8	36358.00	4544.750
16	36435.00	2277.188
32	36407.00	1137.719
64	36680.00	573.125
128	48244.00	376.906
256	71981.00	281.176
512	118923.00	232.271
1024	213591.00	208.585
2048	402990.00	196.772
4096	781795.00	190.868
8192	1539489.00	187.926
16384	3049900.00	186.151
32768	6086248.00	185.738
65536	12122159.00	184.969
131072	24227546.00	184.842
262144	48432636.00	184.756
524288	96865188.00	184.756
1048576	194294919.00	185.294
2097152	193858056.00	92.439
4194304	193951611.00	46.242
8388608	388310426.00	46.290
16777216	775342575.00	46.214
33554432	1551864643.00	46.249
67108864	3102265628.00	46.227
134217728	6204999206.00	46.231
268435456	12409951078.00	46.231
536870912	24819267802.00	46.229

Table 27: Linux64 Parallel C with Constant Time Assembly Code: 512 Bit Hash Length