Design and Primitive Specification for Boole

Gregory G. Rose

ggr@qualcomm.com

QUALCOMM Inc.

5775 Morehouse Drive

San Diego, CA, 92121

USA

Ph: +1-858-651-5733

Fax: +1-858-651-5766

Table of Contents

1 Justification

Boole is a cryptographic primitive that can be used as a hash function, message authentication code (MAC) and a synchronous stream cipher. Boole was designed in response to NIST's Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family[18], herein referred to as SHA-3. While this document describes different size variants and different modes of operation of Boole, the 64-bit version used as a hash function should be considered to be our official submission in response to the request.

Boole is named in memory of George Boole[20] (1815-1864), British mathematician and inventor of a logical calculus now called Boolean Algebra, one of the fundamentals of cryptography and computer science. The name is appropriate as Boole uses mostly Boolean operations. Boole is an expansion of a previous stream cipher, Shannon[7], also influenced by members of the SOBER family of stream ciphers[11][13][19], Trivium[4], Scream[10], and SHA-256[8]. It is designed somewhat in the manner of a cryptographic sponge[21], which work should be consulted for various proofs of security. Its cryptographic state consists of a single word-wide, 16-element nonlinear feedback shift register and three extra accumulator words; supported word sizes are 16, 32 and 64 bits.

The philosophy behind Boole is to have a design that introduces nonlinearity using simple (in hardware) and fast Boolean operations, combined with a register design that quickly piles up these nonlinearities. When data is being input it first goes into accumulators, then into the register; subsequently data from the accumulators will be reintroduced to the register, ensuring that introduced differentials will have long term effects. It is designed to "push the envelope" in speed, and may well be broken during the NIST competition, but it is hoped that the design will at least be interesting to study.

Boole is free to use for any purpose, and reference source code can be found from the NIST submission site[18].

2 Introduction

Boole is a primitive (more correctly a collection of primitives) that operates on W-bit words, $W \in \{16, 32, 64\}$. W is referred to as the word size, and when we need to distinguish between these sizes we will refer to, e.g., Boole64. It can be used as a hash function, a keyed message authentication function, pseudorandom number generator or a stream cipher.[1] Inputs and outputs are generally arbitrary length sequences of bits as defined for SHA-3. As a hash function, Boole supports output lengths of up to $8W$ bits. Thus Boole64 supports all of the required output lengths for SHA-3. For up to 256-bit digests, implementations on 32-bit CPUs, or where resources for storage of state information are somewhat

[1] Unlike some of our previous designs, Boole shall not simultaneously operate as a stream cipher and MAC, however our provided source code (See Appendix A) has been written to allow two instantiations of Boole to work together to appear to accomplish that task. This allows commonality of hardware blocks and/or code, and could take advantage of parallel processing.

constrained, Boole32 might be considered. While Boole16 does not support output lengths for SHA-3, it might be useful as a replacement for MD4/MD5 or a stream cipher and/or MAC for constrained environments, and represents a “reduced size” version for trying out possible attacks.

Boole is a software-oriented primitive based on simple word operations (operations on data are mainly XOR, NOT, OR and fixed rotations and shifts). Consequently, Boole is at home in many computing environments, from simple hardware implementations through smart cards to large computers. Source code for Boole is freely available and use of this source code, or independent implementations, is allowed free for any purpose.

Boole is a back-to-basics design incorporating lessons learned from a variety of sources. From members of the SOBER family of stream ciphers, it gets its basic shift register structure. Trivium showed how a simple nonlinear feedback structure could compound rapidly to provide security, Scream first taught us the value of keeping the nonlinearity in the cipher state. SHA-256, in its resistance to the attacks against earlier hash functions[15], demonstrates the importance of propagating differentials forward for hash functions and message authentication codes. PANAMA and subsequent work on cryptographic sponges give useful insight into using a stream cipher as a hash function. Finally, many aspects of the design have been influenced by the theory of Golomb Rulers[9] (also often known as Full Positive Difference Sets). The use of only extremely primitive operations and no tables follows work by Bernstein[1] on timing attacks related to table lookups. Lastly we would like to credit Bart Preneel and Helena Handschuh for emphasizing the fragility of plaintext-aware stream ciphers.

2.1 Usage and threat model

As a hash function, Boole is intended to be “like a random oracle”, in the spirit of SHA-3. See the request for details of what this means. Boole supports messages up to 2^{64} bits, although it would be easy to design a variation without this limitation.

When operated with a key, Boole offers message encryption or message integrity protection. Our supplied source code uses two instantiations of Boole data structures to achieve both at once. Our code requires use of a per-message nonce when used as a stream cipher and/or MAC. The only requirement imposed on the nonce is uniqueness. Boole is intended to provide security under the condition that no nonce is ever reused with a single key, that no more than 2^{3W} words of data are processed with one key, and that no more than 2^{64} bits of data are processed with one key/nonce pair. There is no requirement that nonces be random; this allows use of a counter, and makes guaranteeing uniqueness much easier.

Since Boole directly supports use of a key for message authentication, it is not necessary for it to be used with constructs such as HMAC.

Keys and nonces can be of any bit length, as they are processed exactly as if they were input to the hash function. The security level of Boole used as a MAC or stream cipher should be equivalent to brute force with up to a 2^{4W}-bit key.

2.2 Formal declarations

The designers state that we have not inserted any deliberate weaknesses, nor are we aware (at the time of writing) of any deficiencies of the primitive that would make it unsuitable for SHA-3.

QUALCOMM Incorporated allows free and unrestricted use of its intellectual property required to exercise the primitive as specified, including use of the provided source code. The designers are unaware of any intellectual property owned by other parties that would impact on the use of Boole. The designers undertake to inform the SHA-3 project of any changes regarding the intellectual property claims covering Boole.

2.3 Outline of this Document

Section 3 contains a description of Boole. An analysis of the security characteristics, and corresponding design rationale of Boole is found in Section 4. Section 5 outlines the strengths and advantages of Boole. Computational efficiency is discussed in Section 6. Appendices provide a recommended C-language interface, tables of initialization values, and an annotated description of the execution of a two-word hash calculation.

2.4 Notation and conventions

$a<<<b$ (resp. >>>) means rotation of the word a left (respectively right) by b bits; note that Boole uses only constants for the rotation amount.

~ is bitwise complement of W-bit words.

$\oplus$ is exclusive-or of W-bit words.

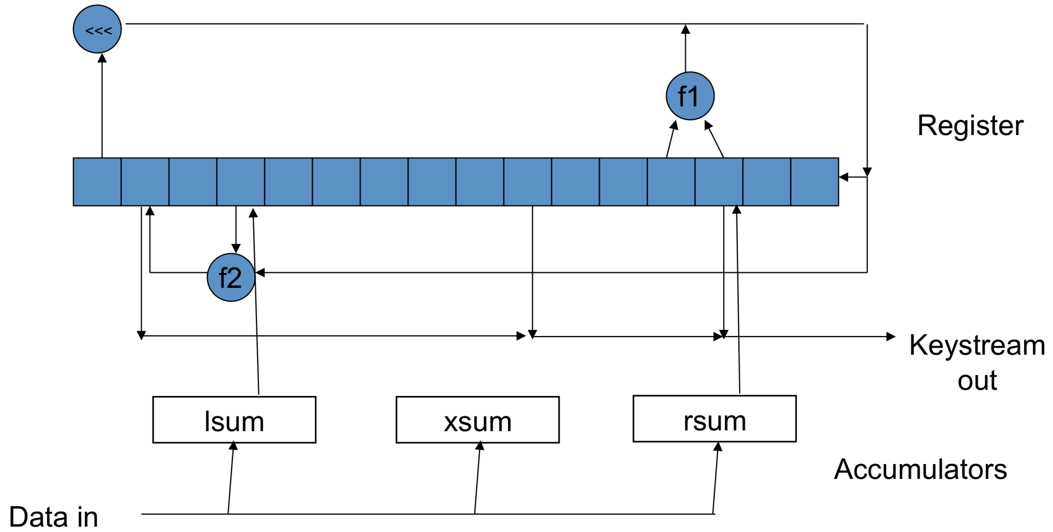
| is bit-wise "or" of W-bit words.

+ is addition modulo 2^W of W-bit words.

Boole is entirely based on W-bit word operations internally (from now on we will just refer to words), but the external interface is specified in terms of arrays of bytes (see SHA-3). A bit sequence of less than a whole byte is represented most significant bit first. Conversion between W-bit chunks and words is done in "little-endian" fashion (as is native on Intel CPUs) irrespective of the byte ordering of the underlying machine. When the byte string is not big enough for a whole word, it is as if there were sufficient zero bytes available. For example, the 9-bit string '111100011' would become the 32-bit word 0x000080F1.

3 Description

Figure 1 -- Schematic of Boole



3.1 Phases

Boole's data structure, shown in Figure 1 -- Schematic of Boole, is constructed from a *non-linear feedback shift register*, *input accumulators* and an *output filter function*. In the figure, elements of the register are shown in the positions "before cycling" (see below). Boole operates in three distinct phases:

- *Input*, where words of data to be hashed, or key or nonce material, are input to the accumulators and the register;
- *Mixing*, in which the length in bits of data previously input, the intended length of any output (zero for operation as a stream cipher), and the accumulators, are mixed into the register, and
- *Output*, where the contents of the register and the output filter function are used to generate output keystream either for use as a stream cipher or as the hash or MAC output.

At the end of any distinct input phase, mixing will occur. There might be multiple input phases. For example, to be used to generate a MAC, the phases will probably be input (key), mixing, input (nonce), mixing, input (data), mixing, output (MAC).

3.2 Register cycle

The primitive is based on W-bit operations and W-bit blocks: each block is called a *word*[2]. The vector of words $\sigma_t = (R_t[0], \dots, R_t[15])$ is known as the *state* of the *register* at time t, and the state $\sigma_0 = (R_0[0], \dots, R_0[15])$ is called the *initial state*, and is used directly in the hash function. The *key state* is initialised from the secret key by the *key loading*. The key state can be used directly as the initial state for message authentication, or can be further perturbed by the nonce loading process to form the initial state for use as a stream cipher or MAC[3].

The state transition function of the register, referred to as a *cycle*, transforms state σ_t into state σ_{t+1} in the following manner:

1. $R_{t+1}[i] \leftarrow R_t[i+1]$, for $i = 1..14$ (that is, the middle words of the register are derived by shifting).

2. $R_{t+1}[15] \leftarrow f_1(R_t[12] \oplus R_t[13]) \oplus (R_t[0] \lll 1)$

3. $R_{t+1}[0] \leftarrow R_{t+1}[0] \oplus f_2(R_{t+1}[2] \oplus R_{t+1}[15])$ (that is, "feed forward" to the new lowest element)

The nonlinear functions f_1 and f_2 are defined in section 3.6 below.

3.3 The accumulators and input phase

There are three word accumulators, updated during the input phase and used during the input and mixing phases. They are called the *left sum*, *XOR sum* and *right sum* (abbreviated *l, x, r* respectively). The function f_1 is defined in section 3.6 below, and is the same function that is used in the register cycling described above. In an input phase, the input bit sequence is formatted into words as described above. The tth word of input w_t is used to update the accumulators and the register before cycling in the following manner:

1. Let $temp = f_1(l_t) \oplus w_t$

2. $l_{t+1} = temp \lll 1$

3. $x_{t+1} = x_t \oplus w_t$

4. $r_{t+1} = (r_t \oplus temp) \ggg 1$

5. $R_t[3] \leftarrow R_t[3] \oplus l_{t+1}$

6. $R_t[13] \leftarrow R_t[13] \oplus r_{t+1}$

---

2 SHA-3 refers to "blocks" in the sense of Merkle-Damgård. Boole does not have this concept. In one sense, the block is a W-bit word, in another it is the entire input. The fast implementation of the provided source code prefers to work in 16-word chunks.

3 The code provided insists on use of a nonce since it is expected that encryption will be used.

7. cycle the register as above.

The purpose of the accumulators is to ensure that any differentials introduced by any input word continue to be introduced into the register until they are explicitly cancelled out by other differences. Since the accumulators have different rotation directions and operations, cancelling their effects is hopefully a complex undertaking.

3.4 Mixing phase

After any input phase, the state is further mixed into the register. First, the length in bits of the input (data, key, nonce), represented as a 64-bit integer, is split into W-bit words, least significant words first, to form $L[i]$, i=0..(64/W – 1). (Obviously this is a null operation when W=64.) Let h be the length of the requested hash or MAC output, and set h to zero for operation as a stream cipher. Mixing is accomplished as follows:

1. $R[0..(64/W - 1)] = R[0..(64/W - 1)] \oplus L[0..(64/W - 1)]$ (that is, XOR the length, least significant word first, into as many of the first four words of the register as are needed for the 64-bit quantity.)

2. $R[4] = R[4] \oplus h \oplus l$

3. $R[i] = R[i] \oplus l, \forall i \in \{7, 10, 13\}$

4. $R[i] = R[i] \oplus x, \forall i \in \{5, 8, 11, 14\}$

5. $R[i] = R[i] \oplus r, \forall i \in \{6, 9, 12, 15\}$

6. cycle the register 16 times.

This procedure is not uniquely invertible, and serves to even further mix the input data into the state of the register. A single mixing phase is performed after entry of the key material, nonce material, and input data when generating a MAC. For reasons of paranoia, the mixing phase is performed twice after input of data for generating a hash.

3.5 Output phase

When output is required for use as keystream, MAC or message digest, the register is cycled and a simple filter function of the register contents is used to generate words of output v as required. In detail:

1. Cycle the register. Note that the accumulators are not used during the output phase.

2. Let $v = R[0] \oplus R[8] \oplus R[12]$

3. Output v for hash or MAC, or XOR v into the output buffer for stream cipher.

3.6 The S-Box Functions f_1 and f_2

There are two related nonlinear functions used in the state updating (and output) function of Boole, mapping an input word to an output word. These functions are intended to be easily and efficiently computable in both software and hardware without using lookup tables, and to simulate a random function. The functions consist basically of ORing rotated copies of the input word, and XORing the result back into the word. This is the only area of Boole that differs depending on word size, as there are slightly different forms of the functions, and the rotation constants also depend on the word size (see below for a discussion of Golomb Rulers and how candidate rotation constants were generated). Both functions have the same form given an input word w, but the rotation constants will differ between the two. The following sections discuss how the constants used in the functions were chosen.

16-bit word size

There were four candidate sets of four rotation constants, of which two had rotations that differed by a multiple of 8 bits. This was deemed undesirable, so the other two sets were used (maximum power-of-two difference of 4). For each configuration of the rotation constants, and with a couple of variations such as the NOT function that is used, the mapping was enumerated, and the best coverage set was selected for each set.

1. Let $t \leftarrow w \oplus ((w \lll A) \mid (w \lll B))$

2. Return $t \oplus ((\sim t \lll C) \mid (t \lll D))$

The only difference between the functions is the constants {A, B, C, D}. For f_1 they are {9, 13, 10, 15} respectively, while for f_2 they are {3, 14, 9, 10} respectively. Each bit of the output word is a fourth-degree function of 9 bits of the input. Thus, such functions have a bias of 2^{-3}.

32-bit word size

The functions and rotation constants in this case are identical to those used in the Shannon stream cipher. There were 5 sets of six rotation constants that had a greatest power-of-two difference of 4, and which included 0 as an element of the sets (used implicitly by not rotating the word itself). Of these, only one also had 1 as a constant, and that is used in the feedback function. Thus the functions use the other four constants. All six possible arrangements were exhaustively tested for coverage; two look like random functions, the other four had undesirably low coverage. Note that the selected functions have zero as a fixed point; this shouldn't matter but in hindsight might have been avoided if we had tested inclusion of the NOT as in the 16-bit case, or a constant as in the 64-bit case.

1. Let $t \leftarrow w \oplus ((w \lll A) \mid (w \lll B))$

2. Return $t \oplus ((t \lll C) \mid (t \lll D))$

The only difference between the functions is the ordering of the constants {A, B, C, D}. For f_1 they are {5, 7, 19, 22} respectively, while for f_2 they are {7, 22, 5, 19} respectively. Each bit of the output word is a fourth-degree function of 9 bits of the input. Thus, such functions have a bias of 2^{-3}.

64-bit word size

There were 187 sets of eight rotation constants in the candidate pool, that included both 0 and 1 in the sets, and which had a greatest power-of-two difference of 8. We empirically tested a number of different configurations of one set of the constants, and decided on the general shape of the function:

1. Let t $\leftarrow w \oplus$ 0x6996c53a
2. Let $t \leftarrow t \oplus ((t <<< C) \mid (t <<< D))$
3. Let $t \leftarrow t \oplus ((t <<< B) \mid (t <<< E))$
4. Return $t \oplus ((t << A) \mid (t <<< F))$

Note that the term with constant A is shifted, not rotated, bringing in zero bits at one end; this breaks the rotational symmetry of the construction. The constant in step 1 ensures that zero is not a fixed point, and it is traditional in constructs derived from SOBER. Empirically, the best results were obtained when the last step's constants were furthest apart, and we wanted to minimize the number of bits discarded in step 4. f_1 uses left rotations and shifts, as shown, while f_2 shifts and rotates right instead.

Evaluation of the candidates by enumeration was infeasible at 64 bits. Instead, a cycle finding program was used to determine the average tail length and cycle length from eight random starting points and also the words consisting of all 0 bits and all 1 bits. The intent was to select functions that had similar characteristics to random functions, that is, had an average tail and cycle length of about $2^{W-1}\sqrt{\pi/2}$, 2 691 471 335. Four candidate sets stood out; the two with the smallest value for A were tested further, averaging 100 starting points, and showed no anomalies, so they were selected.

The differences between the functions is the rotation direction and the set of the constants {A, B, C, D, E, F}. For f_1 they are {3, 20, 34, 42, 55, 60} respectively, while for f_2 they are {5, 27, 35, 46, 52, 55} respectively. Most bits of the output word are an eighth-degree function of 26 bits of the input, however the least significant 3 bits of f_1 and most significant 5 bits of f_2 are only degree 4. Thus, some bits of such functions have a maximum bias of 2^{-3}.

3.7 Initial State, Key and Nonce Loading

The initial state of the shift register and accumulators before input of hash data or key material for different word sizes is given in Appendix B. The random looking values of R are actually generated iteratively using the first of the nonlinear functions:

1. $R_0[0] \leftarrow f_1(1)$

2. $R_0[i] \leftarrow f_1(R_0[i\text{-}1]), \forall i \in \{1..15\}$

The initial values of the accumulators derive from the magic number 0x6996C53A, which has traditionally been used in all SOBER-derived ciphers. Accumulator x is zeroed, l is set to the magic number (truncated to 0xC53A in the 16-bit case), and r is set to the value of l rotated left by 8 bits. Each time a new input phase is to begin, the accumulators are reset to these known values; this occurs before keying, providing a nonce, and data input for hash or MAC.

Boole, used as a stream cipher or MAC, is keyed and re-keyed (that is, incorporating the nonce) by using the key or nonce as data input to the hash function. After the input and mixing phases, the contents of the shift register are used.

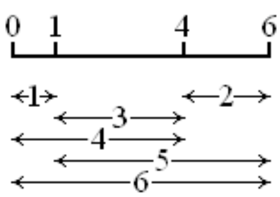
The source code supplied implements integrated stream cipher and MAC usage. It is important for security that these two components start off with independent states, but equally important for ease of use and efficiency that applications only have to maintain one key. When finished loading the key (into the hash function state), the register is copied out so that key loading need not be repeated. The supplied code requires use of a nonce. When generating a MAC, the saved post-keying state is copied into the register, and the nonce is then input as for a hash, and mixed. For independence when generating stream cipher output, the saved post-keying state is copied into the stream cipher register *in reverse order*, and the nonce is then input as for a hash, and mixed. There is no apparent statistical relationship between these two initial states at the end of these processes.

4 Design and Security Analysis of Boole

Many of the components of Boole have been subjected to scrutiny when they appeared in earlier members of the SOBER family of stream ciphers.

4.1 <u>Design using Golomb Rulers</u>

The design of Boole as a nonlinear shift register required three applications of Golomb Rulers, or Full Positive Difference Sets. *In mathematics, a Golomb ruler, named for Solomon W. Golomb, is a set of marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between any two of its marks is its length. Translation and reflection of a Golomb ruler are considered trivial, so the smallest mark is customarily put at 0 and the next mark at the smaller of its two possible values.*



[Description and image courtesy of Wikipedia.]

Boole requires three applications of Golomb Rulers. The first is for the basic nonlinear shift register. Given the selected 16-word length of the register, we require a ruler of length 17 with marks at 0 and 16. Simultaneously, though, we require a ruler for the output and "feed forward" function. This must be a ruler of length at most 16. These two rulers needed to be generated simultaneously and machine optimised for their resistance to "guess and determine" attacks (see, for example, [2], [3]). There are exactly three combinations that required the maximum 11 words of guessing before being able to determine the register. This is equivalent to an effort of 2^{11W}. From these three possibilities, we chose marks of (0, 12, 13, 16) for the main feedback register and (2, 8, 12, 15) for the output function for no particular reason. These design elements appeared first in the Shannon stream cipher, and are unchanged.

We wanted the nonlinearity in Boole to come from a very simple function that transformed a single word using rotations and Boolean operations. It is important for this application to get maximum diffusion as the nonlinear function is "piled up", so we want to apply the concept of a Golomb Ruler to the shift amounts for the 32-bit words. However, rotations "wrap around" so we want to apply the concept "modulo W". The chosen set of rotations is for the various word sizes are discussed above. The constraints were:

1. Should have 0 (because no rotation is no work) and 1 (some embedded CPUs can execute rotations by a single bit more efficiently, and rotation by one is used in the feedback function). This was impossible for the 16-bit case, so we took sets that started at 1.

2. Maximum possible number of marks (5/6/8)

3. Of the possible even differences, we want the set that minimises the greatest power of two that divides any difference. (4/4/8)

We wanted to use the single-bit rotation in the main feedback loop, and the "no rotation" for the base of the nonlinear functions, leaving the other marks for the rotations inside the function.

As mentioned above, the 32-bit case is exactly what was used in Shannon, while the 16- and 64-bit selections were new.

4.2 <u>Security Requirements</u>

Boole as a hash function is hoped to provide security for outputs of up to 2^{8W} bits, according to the SHA-3 criteria.

Boole as a stream cipher is intended to provide 2^{4W}-bit security, matching with the usual security correspondence between hash functions and ciphers. The base attack on the Boole stream cipher is an exhaustive key search, which has a computational complexity equivalent to generating 2^{4W} keystream words[4]. In all attacks, it is assumed that the attacker observes a certain amount of keystream produced by one or more secret keys, and the attacker is assumed to know the corresponding plaintext and nonces. Boole is considered to *resist* an attack if either the attack requires the owner of the secret key(s) to generate more than 2^{3W} key stream words with any one key[5], or the computational complexity of the attack is equivalent to the attacker rekeying the cipher 2^{4W} times and generating at least 5 words of output each time. With respect to the keystream functionality, we claim that Boole fulfils the following security requirements, when used subject to the condition that no key/nonce pair is ever reused, and that no more than 2^{64} words of data are processed with one key/nonce pair:

1. **Key/State Recovery Attacks**: Boole must resist attacks that either determine the secret key, or determine the values of the cipher state at any specified time.

2. **Keystream Recovery Attacks**: Boole must resist attacks that accurately predict unknown values of the keystream without determining information about the shift register state or the secret key.

3. **Distinguishing attacks**: Boole should resist attacks that distinguish a Boole keystream from random bit stream.

4. **Related-Key or related nonce Attacks**: Boole should resist attacks of the above form that use keystreams generated from multiple key/nonce pairs that are related in some manner known to the attacker (including the attacker choosing the nonces).

A separate set of security requirements apply to the MAC functionality. First, we consider the security properties required of a MAC function. A MAC function is a cryptographic algorithm that generates a tag TAG = MAC_K(M) of length *d* from a message M of arbitrary length and a secret key K of length *n*. The message-tag pair (M,TAG) is transmitted to the receiver. The message may be encrypted, in whole or in part, before transmission. Parts of the message that are both encrypted and integrity protected should be encrypted first, that is, the MAC applies to the ciphertext.

Suppose the received message-tag pair is (RM,RTAG). The receiver processes the received message and calculates an expected tag XTAG = MAC_K(RM). If XTAG = RTAG, then the receiver has some confidence that the message-tag pair was formed by a party that knows the key K. Checking the MAC can be done before decryption, or in parallel with it. In most cases, the message includes sequence data (such as a nonce) to prevent replay of message-tag pairs.

The length *n* of the key and the length *d* of the tag form the security parameters of a MAC algorithm, as these values dictate the degree to which the receiver can have confidence that the message-tag pair was formed by a party that knows the key K. A MAC function with security parameters (*n*, *d*) should provide resistance to four classes of attacks:

---

[4] Unless, of course, a shorter secret key is used. We assume use of an appropriate secret key in this section.

[5] This is not a typo; we do not believe that it is reasonable for an attacker to use an amount of keystream that no sensible user would generate.

1. **Collision Attack**. In a collision attack, the attacker finds any two distinct messages M,M' such that $MAC_K(M) = MAC_K(M')$. A MAC function resists a collision attack if the complexity of the attack is $O(2^{\min(n,d/2)})$. Note that meaningful collision attacks against MAC functions are rare in practice.

2. **First Pre-image Attack**. In a first pre-image attack, the attacker is specified a tag value TAG, and the attacker must find a message M for which $MAC_K(M) = TAG$. A MAC function resists a first pre-image attack if the complexity of the attack is $O(2^{\min(n,d)})$.

3. **Second pre-image attack**. In a second pre-image attack, the attacker is specified a message M, and the attacker generates a new message M' such that $MAC_K(M) = MAC_K(M')$. A MAC function resists a second pre-image attack if the complexity of the attack is $O(2^{\min(n,d)})$.

4. **MAC Forgery**. In MAC forgery, the attacker generates a new message-tag pair (M', y') such that $y' = MAC_K(M')$. A MAC function resists MAC forgery if the complexity of the attack is $O(2^{\min(n,d)})$.

In all these attacks, the attacker is presumed to be ignorant of the value of the key K[6]. However, we assume that (prior to the attack) the attacker can specify messages M(i) for which they will be provided with the corresponding tags $TAG(i) = MAC_K(M(i))$.

Boole is intended to be a MAC function with security parameters $n = 2^{4W}$, and $d \le 2^{8W}$.

Boole will be considered broken if an attacker can perform any of these attacks. Keystream recovery attacks seem unlikely, as the output sequence relies heavily on the state of the register, so any likely keystream recovery attack will probably also allow the stronger key/state recovery attack. Most attacks concentrate on the first option of determining the values of the register state. Related-key attacks are of less concern, since most security systems ensure that attackers cannot predict relationships between secret keys. However, it is still preferable that Boole resists such attacks.

A comment on distinguishing attacks. There is currently some debate regarding the complexity of distinguishing attacks on stream ciphers. Some members of the cryptologic community claim that a stream cipher cannot be secure when the data complexity and computational complexity for a successful distinguishing attack is less than the key space. For example, these people would say that Boole is not secure if there is a distinguishing attack requiring 2^{80} key stream words and 2^{100} computations. Other members of the cryptologic community claim that stream ciphers can still be secure when the data complexity and computational complexity for a successful distinguishing attack is less than the limits imposed on other types of attacks. These parties would say that Boole is still secure even if there is a distinguishing attack requiring 2^{64} key stream words and 2^{80} computations. Although

[6] There are circumstances (albeit rare) where the users require a MAC function to resist a collision attack with known key. This is similar to an attack on the hash function, so Boole is intended to prevent this type of attack. We note that some other common MAC constructions, such as CBC-MAC, cannot prevent this type of attack.

the designers hold the second view (that stream ciphers can still be secure even when the complexities of distinguishing attacks fall below the bounds of the keyspace), the intention of the design is to ensure that there are no distinguishing attacks on Boole requiring less than the limits mentioned above. By comparison, AES-256 in counter mode has a distinguishing attack requiring 2^{71} keystream bits.

A comment on nonce reuse. As with any stream cipher, an attacker who can force the reuse of a nonce can easily breach privacy. The attacker might have significant difficulty forcing the sender of a message to do this, but it is generally easy for him to do this to the recipient, who has to process everything that is received. This leads us to believe that plaintext-aware encryption is inherently fragile. Thus (in a departure from our previous designs) Boole requires the use of (and our source code implements) independent states for message integrity and keystream generation.

4.3 Security Claims

We believe that any attack on Boole has a complexity exceeding that of generic attacks (e.g. exhaustive key search, time-memory tradeoffs, etc.) up to 2^{4W}-bit keys. We do not claim any mathematical proof of security. Our analysis of Boole can be summarized thus:

- Guess-and-determine (GD) attacks [2] appear to have a computational complexity well in excess of 2^{11W} (see [2, 12]).
- Algebraic attacks [6] appear to be infeasible due to the rapid accumulation of nonlinearity in the shift register.
- Long-distance correlation-based attacks appear to be resisted by the register nonlinear feedback structure.
- "Crossword puzzle" attacks [5] are infeasible because the bias of the nonlinear functions, "piled up" over the length of the register, is less than 2^{-48}.
- Timing, power, cache timing and branch prediction attacks can be mitigated in standard ways; there is no data-dependent conditional execution during or after initial keying, nor are most of the operations used data-dependent in execution time on most CPUs.
- We are unaware of any ways in which the key loading can be exploited.
- We are unaware of any weak keys or weak-key classes. Note that it is theoretically possible for the initial state to be entirely zero, but this is not relevant with the nonlinear feedback method.
- We have no reason to believe that there are a significant number of cycles in the generated keystream of length less than 2^{80}. Our studies have been unable to demonstrate any cycle, even in Boole16. Algebraic methods for constructing such a cycle have eluded us. Assuming that the nonlinear functions behave well, the expected cycle length is $2^{250/505/1022}$ words. Note that the 16- and 32-bit variants of Boole are rotationally symmetric during keystream generation.

4.4 Heuristic Analysis of Boole

This analysis concentrates on vulnerability of Boole as a stream cipher to known-plaintext attacks. An unknown-plaintext attack on a stream cipher uses statistical abnormalities of the output stream to recover plaintext, or to attack the cipher. Any unknown-plaintext attack would also be manifested as a serious distinguishing attack, so we don't consider this any further.

4.4.1 Analysis of the Hash Function

Recent experience indicates that the easiest attacks against hash functions are against the collision resistance property, so we concentrate our heuristic analysis on that. Collisions in the output of Boole can occur in three ways. Working backward, it is always possible that different shift register states will nonetheless produce identical outputs up to the maximum hash output length allowed. Since the register contains at least twice as much state information as the output hash, there are many sets of such states. Finding preimages for different such states is, by the assumption above, harder than finding other collisions.

It is possible that different states before the mixing phase could result in the same output shift register, and hence a collision. For this to happen, it would be necessary for the shift register and the accumulators to both be different, and the accumulator differences to cancel out the shift register differences. This is why the hash function uses two mixing phases at the end; it would be necessary to find a set of differences that forms a fixed point for the mixing "function". The entire state, consisting of 19 words altogether, would need to be searched to find such fixed points, significantly exceeding the amount of work to find collisions by brute force.

Lastly, then, it seems the most likely source of collisions would be to find distinct inputs that result in the same state (both register and accumulators) before the mixing phases. This is key to the design of Boole. Unlike iterated hash functions, where similar states at the end of one compression function can be corrected in a subsequent one, Boole works in a single, long, compression phase. The attacker has to create his exact collision in one operation. The design using accumulators is meant to make this difficult. Any single-word input difference results in continuously reintroducing variations of that difference into the register. At the end, whatever differences have been introduced are constrained that they must cancel out in all three of the accumulators, and at least the accumulators are easy to analyze; patterns of input that work are severely constrained. But this leaves the attacker with only 2^W opportunities to find collisions before exceeding the brute force workload.

4.4.2 Analysis of the Key Loading

The key loading was designed to ensure that (after all key material has been included), the following properties hold:

- The key length is included to ensure that there are no simply equivalent secret keys or equivalent nonces.

- There is no initial state of the registers that is known to be weak in any sense, so it follows that there are no known weak keys.

We believe that these properties ensure that the key loading cannot be exploited.

4.4.3 Analysis of the stream cipher component.

The "shape" of the state register, in the sense of its feedback taps and nonlinear filter function taps, were mechanically optimized to give maximum resistance to Guess and Determine attacks, which appear to have complexity greater than 2^{8W}.

The feedback function of the stream cipher is somewhat nonlinear, through use of the functions f_1 and f_2. Rotations of the words used in the feedback function inputs ensure that all bits in the register have nonlinear effect on the register contents quite rapidly (within 8 words of output). The nonlinear effects also compound quite rapidly due to the tap and "feed forward" positions. This should be more than ample in practice. In the absence of any reason to believe that the feedback function behaves in a significantly non-random fashion, the average cycle length should be approximately $2^{250/505/1022}$ (for the 16/32/64-bit versions respectively, accounting for the rotational symmetry noted above in the case of 16- and 32-bit versions, and referring to [17]).

The nonlinearity of the feedback function, and the selection of the taps for the output filter function, should adequately disguise any short-distance correlations.

The output filter function is quite simple, and serves mostly to ensure that no exploitable combination of input words appears before many applications of the nonlinear Sbox have been applied.

4.4.4 Analysis of the Boole MAC function

The MAC function's strength primarily relates to the strength of the hash function and the stream cipher combined. Finding the state of the register from MAC output would be a break of the stream cipher, and finding the key from the known state would imply finding at least one preimage, and hence would constitute a break of the hash function.

5 Strengths and Advantages of Boole

Boole has the following strengths and advantages.

- Speed.
- Requires a relatively small amount of memory.
- Flexibility in the processor size and implementation.
- The design allows for the use of a secret key and non-secret nonce.

- Appears to provide more than adequate security.
- Incorporates MAC and stream cipher functionality

6 Performance

Boole is designed to be fast in software and small and fast in hardware. Unlike some contemporary designs, the streaming nature and rapid piling up of nonlinearity precludes any meaningful use of multiple CPUs, although it would be possible to use Boole as a primitive in some kind of tree structure. We find arguments that claim such functionality is necessary are unconvincing. For example, huge gigabit routers can be working on multiple independent packets; hashing entire disk drives will probably be done track-by-track or file-by-file; RFID tags probably won't have multiple cores. Instead, we concentrated on maximum single-CPU throughput. Since operation as a stream cipher with MAC requires two sets of state information, two CPUs (appropriately synchronized) would be advantageous, but we have not attempted to measure this.

6.1 Hardware implementation

We have not implemented Boole in hardware, nor do we have the skills to do so. However its fundamental operations are extremely simple so we can estimate the number of gates required for a naïve implementation. We address only operation as a hash function. The only operation that is not simple is the addition required to count the number of bits of input; since this is independent of virtually all of the other processing and not on the critical path for performance, we assume it would be implemented as a simple 64-bit ripple-carry adder. We assume XOR with a constant, rotations and negations are free, XOR requires three gates and two gate delays, flip-flops require four gates and two gate delays. The numbers shown are for the 64-bit version, although we continue to use the word size for clarity. Smaller versions use fewer operations, and hence fewer gates and delays, for the nonlinear functions.

Function	Number of elements	Type of elements	Gate delays on critical path	NAND-gate multiplier	Total gates
Length counter	64	flip-flop	N/A	4	256
Adder for length	64	full adders	N/A	5	320
Storage for state and input word	$20W$	flip-flop	2	4	5120
Initial values	$19W$	mux	1	2	2432
f_1 on *lsum*	$3W/3W$	OR/XOR	9	1/3	768

XOR on *xsum*	W	XOR	parallel	3	192
XOR on *rsum*	W	XOR	2	3	192
XOR accumulators	$2W$	XOR	2	3	384
f_1 for feedback	$3W/5W$	OR/XOR	13	1/3	1152
f_2 for feedback	$3W/5W$	OR/XOR	13	1/3	1152
Mixing accumulators	$13W$	XOR	N/A	3	832
Output function	$2W$	XOR	N/A	3	384
Total			42		13,184

The table above indicates that a full hardware core for Boole64 should be able to be implemented in about 15K basic gates, and should be able to process 64-bit input words with 42 gate delays, comparable in speed to a 64-bit adder. Boole32 should be implementable in about 9K gates.

6.2 Software implementation

The supplied source code is written in C, and the specifications are in Appendix A below. The reference implementation closely follows the description above.

The fast implementation uses two simple techniques for speed: the shift register is implemented as a circular buffer to avoid moving data unnecessarily; large amounts of data are processed 16 words at a time using constant indexes. In addition, we discovered that the 64-bit Microsoft compiler generated sub-optimal code for the rotate-and-or operation fundamental to the nonlinear functions; by assigning intermediate results to temporary variables, the compiler was convinced to generate code comparable to that of GCC. If the underlying CPU is little-endian the portable code works directly on memory words rather than converting from bytes to words in a portable fashion. We have not used optimized assembler code, but from previous experience feel that this could increase performance another 30-50%, depending on the target platform.

The memory requirements for Boole are reasonable. 16 words are needed for the register, 3 words for the accumulators, and a 64-bit integer counter for the input bits; call this 20 words. By comparison, SHA-512 needs 8 words for the chaining variables and 16 words for the message expansion, along with similar overhead variables. Thus Boole is about 20% smaller state.

The performance figures given were measured on two systems. One was a Windows Vista system configured as specified by SHA-3, booted in either 32- or 64-bit mode as appropriate. The other system is an Apple Macbook Pro, with an Intel Core-2 Duo CPU running at 2.4GHz, with 4 GB of 667-MHz

DDR2 DRAM, using the GCC 4.0.1 compiler provided by Apple. The optimization flag –O3 was specified, and "-m64" for Boole64, "-m32" for the 16- and 32-bit versions. The table below gives the performance figures in cycles per byte of hash input as measured using the "rdtsc" instruction on both machines, for the various sizes of the "fast" implementations. In both cases, a large buffer is 1MB, and the figure should approximate the asymptotic performance, while a small buffer ("Block") is 1600 bytes. The output hash value is $8W$ bits long, the maximum allowed. The column labeled "Win 64/32" is for Boole64 compiled for 32-bit platforms, and demonstrates the advantage of using Boole32 for smaller platforms and hash sizes.

Op/Version	Win 32	Win 64/32	Win 64	gcc 32	gcc 64
Block hash	9.92	21.53	7.68	16.68	8.4
Large hash	8.96	18.32	6.17	15.94	7.07
Stream output	6.17	12.15	4.48	5.51	3.71

The worse performance under gcc for the 16-bit version was because the compiler refuses to issue the 16-bit rotate instructions, and for the 32-bit version does some unnecessary masking operations. Both problems could be solved using assembler for the f functions, and further optimizations would be possible.

The table above includes measurements for stream cipher generation because that is relevant to the mixing and output phases for generating a hash. The mixing phase takes approximately the same time as generating 32 words of stream output. Let M be the length of the input message rounded up to a multiple of W bits, and N be the number of bits of output hash generated, similarly rounded up to a multiple of W bits. Then, using the Windows 64-bit reference platform, the number of cycles to generate the output hash should be approximately $(6.17M/8 + 4.48(256+N/8))$.

We have not measured the performance on any 8-bit platform. The limiting operation on such platforms will certainly be the word rotations. Since the rotations are all by constant amounts, a straightforward implementation would use seven lookup tables to find 16-bit constants to be ORed together to form each rotated byte in turn, thus doing $W/8$ table lookups and OR operations for each rotation. Since the outputs are to be ORed together anyway, the steps in the nonlinear functions will use W/4 table lookups and OR operations, and W/8 XORs per step. Boole64 has 13 rotation operations (two are actually shifts, but the same table could be used for most of the input bytes) per cycle and another 8 for input accumulation. Thus we would expect that the nonlinear functions alone would consume about 110 cycles per byte, and other operations about another 30 c/b. Boole32 would be somewhat more efficient on smaller platforms, principally because there are less steps and hence less rotations in the nonlinear functions.

7 References

1. D. Bernstein, "Cache-timing attacks on AES", http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

2. S. Blackburn, S. Murphy, F. Piper and P. Wild, "A SOBERing Remark". Unpublished report. Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U. K., 1998.

3. C. De Cannière, "Guess and Determine Attack on SOBER", *NESSIE Public Document NES/DOC/SAG/WP5/010/a*, November 2001. See [18].

4. Christophe De Cannière, Bart Preneel. "Trivium - A Stream Cipher Construction Inspired by Block Cipher Design Principles". http://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf .

5. J. Cho, "Crossword Puzzle attack on NLS", submitted to FSE 2007 .

6. N. Courtois and W. Meier, "Algebraic attacks on Stream Ciphers with Linear Feedback", proceedings of *EUROCRYPT 2003*, Warsaw, Poland, May 2003.

7. Design and Primitive Specification for Shannon, IACR EPrint Archive http://eprint.iacr.org/2007/044, P. Hawkes, C. McDonald, M. Paddon, G. Rose and M. Wiggers de Vries.

8. FIPS 180-2 Secure Hash Standard (SHS). See the following web page: http://csrc.nist.gov/CryptoToolkit/tkhash.html.

9. Wikipedia, http://en.wikipedia.org/wiki/Golomb_ruler

10. S. Halevi, D. Coppersmith, C. Jutla, "SCREAM: a software efficient stream cipher", Eprint 2002/19.

11. P. Hawkes and G. Rose. The t-class of SOBER stream ciphers. Technical report, QUALCOMM Australia, 1999. See http://www.qualcomm.com.au.

12. P. Hawkes and G. Rose. Exploiting multiples of the connection polynomial in word-oriented stream ciphers. *Advances in Cryptology - ASIACRYPT 2000, Lecture Notes in Computer Science, vol. 1976, T. Okamoto (Ed.), Springer,* pp. 303-316, 2000.

13. P. Hawkes and G. Rose. "Turing, a Fast Stream Cipher". In T. Johansson, Proceedings of *Fast Software Encryption FSE2003,* LNCS 2887, Springer-Verlag 2003.

14. P. Hawkes, G. Rose. "*Primitive Specification for SOBER-128",* 2003. See eprint.iacr.org/2003/081.pdf.

15. P. Hawkes, M. Paddon and G. Rose. *On Corrective Patterns for the SHA-2 Family,* 2004. See eprint.iacr.org/2004/207.pdf.

16. P. Hawkes, M. Paddon and G. Rose. "*The Mundja Streaming MAC",* 2004. See eprint.iacr.org/2004/271.pdf.

17. A. Menezes, P. Van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

18. Federal Register,Vol. 72, No. 212. See also http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/index.html

19. G. Rose, "A Stream Cipher based on Linear Feedback over $GF(2^8)$", in C. Boyd, Editor, *Proc. Australian Conference on Information Security and Privacy*, Springer-Verlag 1998.

20. http://en.wikipedia.org/wiki/George_Boole

21. Cryptographic Sponges, Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche, http://sponge.noekeon.org/

8 Appendix A: Recommended C-language interface

8.1 Possible return values

As required by SHA-3, the hash, MAC and stream cipher interfaces all return a status code. These are the possible return values:

```
typedef enum {
    SUCCESS = 0,
    FAIL = 1,
    WARN_HASHBITLEN = 2, /* warning when length greater than expected security */
    BAD_TERMINATION = 3, /* update called after odd-sized call to update */
    BAD_NEEDNONCE = 4, /* attempt to use MAC/stream without nonce */
} HashReturn;
```

The generic value *FAIL* is never returned, rather a more specific failure value will be returned.

WARN_HASHBITLEN is returned whenever the requested *hashbitlen* would exceed the security bound ($8W$ bits), or is zero or negative. This is a warning in the sense that the data structure is correctly initialized, and subsequent operations leading to generation of a hash or MAC output will succeed. In particular, if the all-in-one "Hash" function is called with a *hashbitlen* that is merely too long, it will nevertheless have generated the requested hash.

BAD_TERMINATION is returned if a subsequent call to any of the functions other than *Final* or *ble_mac* is made, after a call that specified a buffer length not divisible by 8 bits. It was deemed that the complications and inefficiency of coding for such odd sizes are simply not worth it.

BAD_NEEDNONCE is returned from the keyed functions if a call to *ble_nonce* is needed (that is, after keying or after finishing processing a packet) before calling this function. This is never returned from the hash functions.

8.2 Hash function interface

The recommended data structures and interface for the Boole hash function conforms to the requirements of SHA-3. The type "WORD" must be defined as an unsigned integer type with WORDSIZE bits of precision, for the supported word sizes {16, 32, 64} bits, as required.

```
#define N 16 /* number of register elements, and diffusion iterations */
typedef appropriate_type WORD;
typedef struct {
    int            hashbitlen;    /* number of bits of output -- 0 for stream */
    DataLength     nbits;         /* bits of input seen so far */
    WORD           R[N];          /* Working storage for the shift register */
    WORD           xsum;          /* XOR sum of input words */
    WORD           lsum;          /* rotating addition sum of input words */
    WORD           rsum;          /* rotating addition sum of input words */
    /* the following handle non-whole-word input/output */
    WORD           bbuf;          /* partial word buffer */
    int            nbuf;          /* number of part-word bits buffered */
    /* the following used for circular buffer fast implementation only */
    int            z;             /* current zero position */
} hashState;
```

This structure is used also for generating MACs and stream cipher output.

We reiterate the SHA-3 interface functions here for reference:

```
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);
```

8.3 Stream cipher and MAC interface

Boole can also be used as a stream cipher and MAC. In the reference code provided, this is achieved using a "combo" data structure that includes two copies of the *hashState* structure; one is for accumulating and generating a MAC, the other for stream cipher output. For convenience, these are initialized from the same key, and the state (of the hash structure) is saved after keying to avoid recomputation, however the operation to set a nonce value for a packet initializes the two structures differently.

```
typedef struct {
    hashState     h;              /* includes state for hash/MAC */
    hashState     s;              /* state for stream cipher only */
    WORD          initR[N];       /* copy of post-key register to avoid rekey */
    int           neednonce;      /* nonce must be called */
} ble_ctx;
```

```
HashReturn ble_key(ble_ctx *c, const UCHAR key[], int keylen, int maclen);     /* set key */
HashReturn ble_nonce(ble_ctx *c, const UCHAR nonce[], int nlen);               /* set Init Vector */
HashReturn ble_stream(ble_ctx *c, UCHAR *buf, int nbits);                      /* stream cipher */
HashReturn ble_macdata(ble_ctx *c, UCHAR *buf, int nbits);                     /* accumulate MAC */
HashReturn ble_encrypt(ble_ctx *c, UCHAR *buf, int nbits);                     /* enc+MAC */
HashReturn ble_decrypt(ble_ctx *c, UCHAR *buf, int nbits);                     /* dec+MAC */
HashReturn ble_mac(ble_ctx *c, BitSequence *hashval);                          /* finalize MAC */
```

For all uses, *ble_key* should be called to initialize the data structure, do the equivalent of key scheduling, and specify the expected MAC length. If no MAC is to be generated, zero can be used as a MAC length, although *WARN_HASHBITLEN* will be returned. No interface is provided that does not require use of a nonce per packet. The communication should be broken into messages, and *ble_nonce* should be called at the beginning of each message. Nonces should never be reused, but nonces are otherwise opaque to the system, and could easily be based on counters, timestamps, or whatever.

For all uses of a stream cipher, it is highly recommended to protect the transmission with a MAC. It is also common for a packet to be transmitted partly as plaintext, but with integrity protection covering even the plaintext part. The provided source code supports arbitrary interleaving of calls to *ble_macdata* (for data to be transmitted as plaintext) with calls to either *ble_encrypt* or *ble_decrypt* as appropriate. The interleaving at the receiver's end should match that of the sender for correct operation. The combined calls for encryption and decryption always input ciphertext to the MAC computation. Finally, *ble_mac* should be called to generate a MAC to be transmitted or checked against the received MAC. Because it is always ciphertext that is used to generate the MAC, and the two data structures are separate, it is possible for the receiver to use *ble_macdata* followed by *ble_mac* to verify the received message, before using *ble_stream* to decrypt the appropriate portions. (Personally, we believe that this optimization is unnecessary, since the point of having the MAC is to make it unprofitable for the attacker to generate packets that would make this optimization worthwhile, but there it is.) Lastly, encryption without MAC can be done with *ble_stream*, but we do not recommend this.

Our reference implementation for these primitives provides a bit-wise interface, with the restriction that strange buffer lengths (*nbits* not divisible by 8) can only be used at the end of a packet. If an output hash or MAC is of a strange length, the unused bits at the end of the buffer will be zeroed.

The provided distribution of Boole includes a self-test and timing harness that can be used to find the various cycles/byte figures quoted.

9 Appendix B: Initialization values

For implementations that wish to initialize the *hashState* structure with constants rather than determine them computationally, these are the initial values (in hexadecimal) for the various word sizes. The accumulator *x* (*xsum*) is always initialized to zero.

16-bit case:
lsum = C53A
rsum = 3AC5
R = { d976, f9ac, b534, fd27, 9925, d16c, fd36, dd3c, d5e6, 95a7, d547, f59b, f796, d797, 7716, c404}.

32-bit case:
lsum = 6996c53a
rsum = 96c53a69
R = { 2d4800a1 25fc4641 25a97908 37e059a9 ae7953e9 a62450a1 4731a3a7 0aabab7b f8a9ab69 18e32961 9aedf87b 182dab33 b83baa0b 5863100b 9ab27ba9 327af139}.

64-bit case:
lsum = 000000006996c53a
rsum = 0000006996c53a00
R = { 8eba5fa3a506e0dd 0c0f098e577f425f cf86b7dd66cf5c7f d06753a616f073bf 1db9cc1da2bc2bec 233625573d354832 3dc12be59adcd001 39e8d67ef4547ce2 b06f0b296ba9c939 c88b22b9e79ba7a9 7c3188d2b08b2259 a0e6342a27c6aeb5 95d3a1bd45993191 7840de822bcd4e43 d3e1cb37a11e16f3 54812e5d068d08b4}.

10 Appendix C: Detailed examination of a hash computation

The call for SHA-3 requires details of internal states for computations of the hash function for various output sizes. We include files in the submission to satisfy this requirement, but in unedited format without explanation. Here we show in detail and explain the operations performed by Boole64 computing the 224-bit hash of the input “abcdefghi”. This input forms two words. All values shown are 64-bit words in hexadecimal. At each step, we show the state in the following form:

```
---input-word--- -l-accumulator-- -x-accumulator-- -r-accumulator--
    -------R0------- -------R1------- -------R2------- -------R3-------
    -------R4------- -------R5------- -------R5------- -------R7-------
    -------R8------- -------R9------- -------R10------ -------R11------
```

-------R12------ -------R13------ -------R14------ -------R15------

---output-word--

If there is no applicable input or output word, it will be shown as dashes. Where words have changed since the previous output (other than simply being moved in the register) they will be shown in boldface type.

First, the accumulators and registers are initialized:

---------------- 000000006996c53a 0000000000000000 0000006996c53a00

8eba5fa3a506e0dd 0c0f098e577f425f cf86b7dd66cf5c7f d06753a616f073bf

1db9cc1da2bc2bec 233625573d354832 3dc12be59adcd001 39e8d67ef4547ce2

b06f0b296ba9c939 c88b22b9e79ba7a9 7c3188d2b08b2259 a0e6342a27c6aeb5

95d3a1bd45993191 7840de822bcd4e43 d3e1cb37a11e16f3 54812e5d068d08b4

The first 8 characters of the string are formed into a word, least significant byte first, and input into the register.

6867666564636261 **d0cecccac8c6c4c2** **6867666564636261** **b433b30679532c30**

a34b8695419823d2 cf86b7dd66cf5c7f **00a99f6cde36b77d** 1db9cc1da2bc2bec

233625573d354832 3dc12be59adcd001 39e8d67ef4547ce2 b06f0b296ba9c939

c88b22b9e79ba7a9 7c3188d2b08b2259 a0e6342a27c6aeb5 95d3a1bd45993191

cc736d84529e6273 d3e1cb37a11e16f3 54812e5d068d08b4 **8852b26d5eb4b28a**

The next, and last, character of the string is formed into a word, least significant byte first, and input into the register.

0000000000000069 **a9c342cb22da73ba** 68676665646362**08** **f0690931f41f0af6**

0866bb1c83a72145 00a99f6cde36b77d **b47a8ed680665856** 233625573d354832

3dc12be59adcd001 39e8d67ef4547ce2 b06f0b296ba9c939 c88b22b9e79ba7a9

7c3188d2b08b2259 a0e6342a27c6aeb5 95d3a1bd45993191 cc736d84529e6273

2388c20655011c05 54812e5d068d08b4 8852b26d5eb4b28a **1b943c9130ed4764**

Since that was the end of the input data, the next step is mixing the accumulators and other data into the register. The accumulator words will not change any more but are shown for consistency. The length of the input, 72 bits (hex 0x48) is XORed into R0, the desired output length of 224 bits (hex 0xE0) is XORed into R4, and copies of the accumulators are XORed into R4..R15.

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6

0866bb1c83a721**0d** 00a99f6cde36b77d b47a8ed680665856 233625573d354832

9402692eb806a35b **518fb01b90371eea** **400602189fb6c3cf** **61486072c541d413**

1456eeb7d4e84051 **508f3d1bd3d9a443** **3c10e3766743422b** **a4140be136fd007b**

d3e1cb37a11e16f3 **fd426c9624577b0e** **e035d4083ad7d082** **ebfd35a0c4f24d92**

```
----------------
```

There are now 16 cycles of the register to mix the data thoroughly. Since there is no input, only the first and last words of the register change at each cycle.

```
---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    1622ef0b8ad89cf6 b47a8ed680665856 233625573d354832 9402692eb806a35b
    518fb01b90371eea 400602189fb6c3cf 61486072c541d413 1456eeb7d4e84051
    508f3d1bd3d9a443 3c10e3766743422b a4140be136fd007b d3e1cb37a11e16f3
    fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    1d174d8590f0b18b 233625573d354832 9402692eb806a35b 518fb01b90371eea
    400602189fb6c3cf 61486072c541d413 1456eeb7d4e84051 508f3d1bd3d9a443
    3c10e3766743422b a4140be136fd007b d3e1cb37a11e16f3 fd426c9624577b0e
    e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    f592398fa66635e5 9402692eb806a35b 518fb01b90371eea 400602189fb6c3cf
    61486072c541d413 1456eeb7d4e84051 508f3d1bd3d9a443 3c10e3766743422b
    a4140be136fd007b d3e1cb37a11e16f3 fd426c9624577b0e e035d4083ad7d082
    ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    67b3dd2fef738069 518fb01b90371eea 400602189fb6c3cf 61486072c541d413
    1456eeb7d4e84051 508f3d1bd3d9a443 3c10e3766743422b a4140be136fd007b
    d3e1cb37a11e16f3 fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92
    8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    49916bc342f7aa37 400602189fb6c3cf 61486072c541d413 1456eeb7d4e84051
    508f3d1bd3d9a443 3c10e3766743422b a4140be136fd007b d3e1cb37a11e16f3
    fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb
    34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    86d096a0e4116917 61486072c541d413 1456eeb7d4e84051 508f3d1bd3d9a443
    3c10e3766743422b a4140be136fd007b d3e1cb37a11e16f3 fd426c9624577b0e
    e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1
```

e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28 **9dfcba56297ba007**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
b6025e1a539a0f51 1456eeb7d4e84051 508f3d1bd3d9a443 3c10e3766743422b
a4140be136fd007b d3e1cb37a11e16f3 fd426c9624577b0e e035d4083ad7d082
ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7
a673097a59fb996f 3fe9b7e901d25b28 9dfcba56297ba007 **b8ca589df9f9874e**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
4d4035e902e8db26 508f3d1bd3d9a443 3c10e3766743422b a4140be136fd007b
d3e1cb37a11e16f3 fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92
8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f
3fe9b7e901d25b28 9dfcba56297ba007 b8ca589df9f9874e **2b19b551ec11eb10**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
e481d6d3f85d6dbb 3c10e3766743422b a4140be136fd007b d3e1cb37a11e16f3
fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb
34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28
9dfcba56297ba007 b8ca589df9f9874e 2b19b551ec11eb10 **4ceb10b7f86ee3a2**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
bcc8172672a518af a4140be136fd007b d3e1cb37a11e16f3 fd426c9624577b0e
e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1
e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28 9dfcba56297ba007
b8ca589df9f9874e 2b19b551ec11eb10 4ceb10b7f86ee3a2 **1ec25358a659ea62**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
7d36c17f546f42c5 d3e1cb37a11e16f3 fd426c9624577b0e e035d4083ad7d082
ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7
a673097a59fb996f 3fe9b7e901d25b28 9dfcba56297ba007 b8ca589df9f9874e
2b19b551ec11eb10 4ceb10b7f86ee3a2 1ec25358a659ea62 **46f0fd14c0e302d0**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
c23889936641ddf1 fd426c9624577b0e e035d4083ad7d082 ebfd35a0c4f24d92
8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f
3fe9b7e901d25b28 9dfcba56297ba007 b8ca589df9f9874e 2b19b551ec11eb10
4ceb10b7f86ee3a2 1ec25358a659ea62 46f0fd14c0e302d0 **ab0fccf54bef3474**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
7a4bd6d0c285651f e035d4083ad7d082 ebfd35a0c4f24d92 8fe37b58455c95cb
34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28
9dfcba56297ba007 b8ca589df9f9874e 2b19b551ec11eb10 4ceb10b7f86ee3a2
1ec25358a659ea62 46f0fd14c0e302d0 ab0fccf54bef3474 **747ffb20fa7f68e3**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
5fb4ba25acfb2c65 ebfd35a0c4f24d92 8fe37b58455c95cb 34c63c0207529ed1
e685fe2dc95568e7 a673097a59fb996f 3fe9b7e901d25b28 9dfcba56297ba007
b8ca589df9f9874e 2b19b551ec11eb10 4ceb10b7f86ee3a2 1ec25358a659ea62
46f0fd14c0e302d0 ab0fccf54bef3474 747ffb20fa7f68e3 **391a5d146cc84abe**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
5b1559b531be21f0 8fe37b58455c95cb 34c63c0207529ed1 e685fe2dc95568e7
a673097a59fb996f 3fe9b7e901d25b28 9dfcba56297ba007 b8ca589df9f9874e
2b19b551ec11eb10 4ceb10b7f86ee3a2 1ec25358a659ea62 46f0fd14c0e302d0
ab0fccf54bef3474 747ffb20fa7f68e3 391a5d146cc84abe **af65d76c9c72233d**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
bde6d6f4bff0a1fb 34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f
3fe9b7e901d25b28 9dfcba56297ba007 b8ca589df9f9874e 2b19b551ec11eb10
4ceb10b7f86ee3a2 1ec25358a659ea62 46f0fd14c0e302d0 ab0fccf54bef3474
747ffb20fa7f68e3 391a5d146cc84abe af65d76c9c72233d **2339534bdc80027e**

Again, we repeat mixing the accumulators and other data into the register. The length of the input, 72 bits (hex 0x48) is again XORed into R0, the desired output length of 224 bits (hex 0xE0) is XORed into R4, and copies of the accumulators are XORed into R4..R15.

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
bde6d6f4bff0a1**b3** 34c63c0207529ed1 e685fe2dc95568e7 a673097a59fb996f
962af52223082872 f59bdc334d18c20f 48a351ac0de68db8 82daf79acecb98aa
248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c
8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888

Another 16 cycles of the register...

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6

527395bad6d247fc e685fe2dc95568e7 a673097a59fb996f 962af52223082872 f59bdc334d18c20f 48a351ac0de68db8 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 **d42a3d9c4daaaece**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **39506a4e33430405** a673097a59fb996f 962af52223082872 f59bdc334d18c20f 48a351ac0de68db8 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece **0aebd20307e3f163**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **c01fe2a4af93eb57** 962af52223082872 f59bdc334d18c20f 48a351ac0de68db8 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 **1080c06b9eded773**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **467261a3afbf376a** f59bdc334d18c20f 48a351ac0de68db8 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773 **06c08e60847a78ed**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **b786a19da444325b** 48a351ac0de68db8 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed **6eb4719a834eccbe**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **8c81627f8abf1d2f** 82daf79acecb98aa 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe **9fc9fc3a768e2c49**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6 **28ee43f6e726b1cf** 248c76d29c0d81aa eeab5a695246e094 ef33bfdfe239716a

c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904 c702b109f8114135
d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773
06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49 **dc228ca8bf17bd16**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
85d2aef8b338e050 eeab5a695246e094 ef33bfdfe239716a c368aa902f8c567c
8416f2110e606215 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888
d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed
6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16 **9c7a7c140b44f9e7**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
28b46023bdf0fb5e ef33bfdfe239716a c368aa902f8c567c 8416f2110e606215
90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece
0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe
9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7 **c38ed2ed91c35749**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
3a3c03239f1d4447 c368aa902f8c567c 8416f2110e606215 90d91fdf4e123904
c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163
1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49
dc228ca8bf17bd16 9c7a7c140b44f9e7 c38ed2ed91c35749 **b3df4da6f4056f00**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
431e696898d9e596 8416f2110e606215 90d91fdf4e123904 c702b109f8114135
d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773
06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16
9c7a7c140b44f9e7 c38ed2ed91c35749 b3df4da6f4056f00 **e289fdce841c3313**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
acd65ca1efe867c6 90d91fdf4e123904 c702b109f8114135 d3505a7a289f0888
d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed
6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7
c38ed2ed91c35749 b3df4da6f4056f00 e289fdce841c3313 **5631b50004597490**

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
d629283b2d01881e c702b109f8114135 d3505a7a289f0888 d42a3d9c4daaaece
0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe

```
    9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7 c38ed2ed91c35749
    b3df4da6f4056f00 e289fdce841c3313 5631b50004597490 93bb93c3adc70fdf
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    1f844f907ddf3f6c d3505a7a289f0888 d42a3d9c4daaaece 0aebd20307e3f163
    1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49
    dc228ca8bf17bd16 9c7a7c140b44f9e7 c38ed2ed91c35749 b3df4da6f4056f00
    e289fdce841c3313 5631b50004597490 93bb93c3adc70fdf 5b520f8abf1ae9f8
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    7549a019ed89ab29 d42a3d9c4daaaece 0aebd20307e3f163 1080c06b9eded773
    06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16
    9c7a7c140b44f9e7 c38ed2ed91c35749 b3df4da6f4056f00 e289fdce841c3313
    5631b50004597490 93bb93c3adc70fdf 5b520f8abf1ae9f8 4c8c3d9d470e8b47
----------------

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    eb808df67c769ff4 0aebd20307e3f163 1080c06b9eded773 06c08e60847a78ed
    6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7
    c38ed2ed91c35749 b3df4da6f4056f00 e289fdce841c3313 5631b50004597490
    93bb93c3adc70fdf 5b520f8abf1ae9f8 4c8c3d9d470e8b47 456c95a61429ff4b
----------------
```

That finishes the processing of the input. Now the register is used as the basis of a stream cipher, cycling and then producing a word of output. Each output word is the XOR of three register words, shown in italic type.

```
---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    4654a2cb7864c0d9 1080c06b9eded773 06c08e60847a78ed 6eb4719a834eccbe
    9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7 c38ed2ed91c35749
    b3df4da6f4056f00 e289fdce841c3313 5631b50004597490 93bb93c3adc70fdf
    5b520f8abf1ae9f8 4c8c3d9d470e8b47 456c95a61429ff4b 7c447dfa6582539e
aed9e0e7337b4621

---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6
    4616ac51d44a3289 06c08e60847a78ed 6eb4719a834eccbe 9fc9fc3a768e2c49
    dc228ca8bf17bd16 9c7a7c140b44f9e7 c38ed2ed91c35749 b3df4da6f4056f00
    e289fdce841c3313 5631b50004597490 93bb93c3adc70fdf 5b520f8abf1ae9f8
    4c8c3d9d470e8b47 456c95a61429ff4b 7c447dfa6582539e c739f3d25e3aba39
e8136c0217588add
```

`---------------- aa9c342cb22da73ba 6867666564636208 f0690931f41f0af6`

`1dc7fda2620b245f` `6eb4719a834eccbe 9fc9fc3a768e2c49 dc228ca8bf17bd16`

`9c7a7c140b44f9e7 c38ed2ed91c35749 b3df4da6f4056f00 e289fdce841c3313`

`5631b50004597490` `93bb93c3adc70fdf 5b520f8abf1ae9f8 4c8c3d9d470e8b47`

`456c95a61429ff4b` `7c447dfa6582539e c739f3d25e3aba39` **`fd8d00a77ca438de`**

`0e9add04727baf84`

`---------------- a9c342cb22da73ba 6867666564636208 f0690931f41f0af6`

`a73ad2c978a19ed4` `9fc9fc3a768e2c49 dc228ca8bf17bd16 9c7a7c140b44f9e7`

`c38ed2ed91c35749 b3df4da6f4056f00 e289fdce841c3313 5631b50004597490`

`93bb93c3adc70fdf` `5b520f8abf1ae9f8 4c8c3d9d470e8b47 456c95a61429ff4b`

`7c447dfa6582539e` `c739f3d25e3aba39 fd8d00a77ca438de` **`330819d25119a0aa`**

`48c53cf0b0e4c295`

Once the output words are represented as bytes, least significant byte first, we have produced the output hash (shown as hexadecimal bytes). Half of the last output word is unused and discarded.

```
21 46 7b 33 e7 e0 d9 ae dd 8a 58 17 02 6c 13 e8 84 af 7b 72 04 dd 9a 0e 95 c2 e4 b0
```