The SHAvite-3 Hash Function

Eli Biham[1,⋆] and Orr Dunkelman[2,⋆⋆]

[1] Computer Science Department, Technion
Haifa 32000, Israel
biham@cs.technion.ac.il
[2] École Normale Supérieure
Département d'Informatique,
CNRS, INRIA
45 rue d'Ulm, 75230 Paris, France
orr.dunkelman@ens.fr

Abstract. In this document we present SHAvite-3, a secure and efficient hash function based on the HAIFA construction and the AES building blocks. SHAvite-3 uses a well understood set of primitives such a Feistel block cipher which iterates a round function based on the AES round. SHAvite-3's compression functions are secure against cryptanalysis, while the selected mode of iteration offers maximal security against black box attacks on the hash function. SHAvite-3 is both fast and resource-efficient, making it suitable for a wide range of environments, ranging from 8-bit platforms to 64-bit platforms (and beyond).

1 Introduction

The recent security findings on the (lack of) collision resistance in SHA-1 [23, 55] mark the close end of SHA-1's useful life. Although the use of SHA-256 may be a solution for this specific issue, the recent collision finding techniques as well as the results on the second preimage resistance of Merkle-Damgård hash functions and the similarity of the SHA-256 design to the design of SHA-1, motivated the US National Institutes of Standards and Technology to issue a call for a successor algorithm to be named SHA-3. The essential requirements for SHA-3 are the support for message digests of 224, 256, 384, and 512 bits.

In this document, we present a candidate for SHA-3. Our design philosophy is to use well-understood components to achieve high security and competitive performance. We find this approach the most reasonable one given the advances in cryptanalysis of hash functions, and specifically, the results on SHA-1 and on Merkle-Damgård hash functions.

A hash function is usually composed of a compression function and a mode that iterates this compression function to deal with arbitrarily long messages. For the compression function we developed a construction based on the well understood Davies-Meyer transformation of a block cipher into a compression function. The underlying block cipher is a Feistel construction which uses the AES round as a building block.

The hash function then iterates the compression function using the HAsh Iterative FrAmework (HAIFA). The result is a fast and secure hash functions, which can be used to produce any digest size up to 512 bits. For digests of up to 256 bits we allow messages of up to $(2^{64} - 1)$

⋆ The first author was supported in part by the Israel MOD Research and Technology Unit.
⋆⋆ The second author was supported by the France Telecom Chaire.

bits, and for longer digests we allow messages of up to $(2^{128} - 1)$ bits in line with the current FIPS tradition of SHA-1 and the SHA-2 family, ensuring easy transition to SHAvite-3.

As SHAvite-3 is based on AES building blocks, as well as the HAIFA mode of iteration, it is assured to be compact and efficient, and suitable to many platforms (both modern CPUs, as well as smart card and 8-bit machines). Our current implementation of SHAvite-3 achieves for 256-bit digests a speed of 35.3 cycles per byte on a 32-bit machine and of 26.7 cycles per byte on a 64-bit machine. For 512-bit digests, SHAvite-3 achieves speeds of 58.4 cycles per byte on a 32-bit machine, and 38.2 cycles per byte on a 64-bit machine.

SHAvite-3 is named after its speed and security, as it is both a secure hash function, and fast (vite in French). In Hebrew, the meaning of the word shavite is comet, a fast natural phenomena. The current proposed version is SHAvite-3 (pronounced "shavite shalosh", as in Hebrew), as it is the third variant of the design (the first two are unpublished).

This document is organized as follows: In Section 2 we describe the AES round function and some mathematical background related to it. Section 3 outlines HAIFA which is the way SHAvite-3 iterates its compression function. The full specifications of SHAvite-3 are given in Section 4. The design criteria and motivation is outline in Section 5. The security analysis is detailed in Section 6, and we introduce an efficient MAC based on SHAvite-3 in Section 7. We present our performance analysis in Section 8. Several test vectors are given in Appendix A, and detailed internal values during the execution of SHAvite-3 for several messages is given in Appendix B. We summarize the proposal in Section 9.

2 AES and Some Mathematical Background

Our construction relies on the round function used in AES [51]. The advanced encryption standard is an SP-network with block size of 128 bits which supports key sizes of 128, 192, and 256 bits. A 128-bit plaintext is treated as a byte matrix of size 4x4, where each byte represents a value in $GF(2^8)$. An AES round applies four operations to the state matrix:

- SubBytes (SB) — applying the same 8-bit to 8-bit invertible S-box 16 times in parallel on each byte of the state,
- ShiftRows (SR) — cyclic shift of each row (the i'th row is shifted by i bytes to the left),
- MixColumns (MC) — multiplication of each column by a constant 4x4 matrix over the field $GF(2^8)$, and
- AddRoundKey (ARK) — XORing the state with a 128-bit subkey.

We outline an AES round in Figure 1. We note that we only use the full round function of AES, and thus we omit here the full description of the key schedule and the exact definition of AES.

Throughout this document we denote by $AESRound_{subkey}(x)$ one round of AES as defined in the FIPS [51], using the subkey *subkey* applied to the input x. Specifically,

$$AESRound_{subkey}(x) = MC(SR(SB(x))) \oplus subkey.$$

In AES, each byte represents a value in the field $GF(2^8)$, i.e., the byte value $0x13$ corresponds to the polynomial $x^4 + x + 1$. In order to explicitly the AES designers picked the following irreducible polynomial used to generate this field:

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

Thus, to multiply two elements $p(x), q(x) \in GF(2)/m(x)$ (which we denote by $\bullet$, following the Federal Information Processing Standard 197 [51]), first compute the product of $p(x)q(x)$

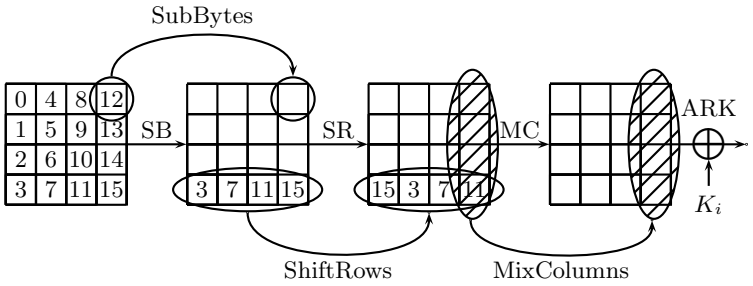


Fig. 1. An AES round

(as polynomials over $GF(2)$), and then reduce the outcome modulo $m(x)$. For example, let $p(x) = x^6 + x^5 + x^2 + 1$ (i.e., $p(x)$ represents the value 65_x) and let $q(x) = x^7 + x^3 + x$ (i.e., $q(x)$ represents the value $8A_x$), then $p(x) \bullet q(x) = x^7 + x^4 + x^3 + x^2 + 1$ (i.e., corresponding to $9D_x$) as

$$p(x)q(x) = \left[x^6 + x^5 + x^2 + 1\right] \cdot \left[x^7 + x^3 + x\right] = x^{13} + x^{12} + x^8 + x^6 + x^5 + x$$

which reduces to $x^7 + x^4 + x^3 + x^2 + 1$ modulo $m(x) = x^8 + x^4 + x^3 + x + 1$.

The MixColumns operation takes each 4-byte column $(b_0, b_1, b_2, b_3)^T$, and multiplies it (from the left) with an MDS matrix over the field $GF(2^8)$, thus the output $(d_0, d_1, d_2, d_3)^T$ is computed as

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

The computation of the S-box is done as follows:

- Given the input x, compute $r = x^{-1}$ in the field GF(2^8) (where zero is considered its own inverse).
- Compute $y = A \cdot r + b$ as linear equations over $GF(2)$ where

$$A = \begin{pmatrix} 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

- Output y.

For completeness, we provide the S-box in Table 1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
10	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
20	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
30	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
40	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
50	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
60	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
70	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
80	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
90	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A0	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B0	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D0	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E0	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F0	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 1. AES' S-box (all values are given in hexadecimal)

3 HAsh Iterative FrAmework

The most widely used mode of iteration is the Merkle-Damgård construction [22, 42, 43]. While the collision resistance of the compression function is preserved in the Merkle-Damgård construction, this is not the case for second preimage resistance as suggested in [1, 25, 37]. Other undesired properties in such iteration were also suggested: extensions attacks (which lead to some differentiability results [19]) and chosen target preimage attacks [36].

The HAsh Iterative FrAmework allows to overcome these problems while maintaining a simple construction and at the same time allowing for a more flexible hash function (for example, it contains an integrated support for variable digest length). Under reasonable assumptions, it is claimed that HAIFA does preserve the major security notions and is indeed second preimage resistant [17]. In particular, we show that the HAIFA mode of iteration is protected against the second preimage attacks of [1, 25, 36, 37].

Moreover, HAIFA has an integrated support for keys, thus defining families of hash functions (when needed). This can also be used as a base for more efficient message authentication codes based on the hash function (as we define in Section 7).

3.1 Specifications of HAIFA

Hashing with HAIFA involves few steps:

1. Message padding, according to the HAIFA padding scheme.
2. Compressing the message using HAIFA-compatible compression function.
3. Truncating the output to the required length.

The message padding used in HAIFA is very similar to the one used in Merkle-Damgård, but offers a better security, as well as better support for different digest sizes. The compression is done using a compression function with four inputs:

- A chaining value (of length m_c),
- A Message block (of length n),
- The number of bits hashed so far including the current block (a counter of length c),

– A salt (of length s).

Hence, to compress a message M, the user first chooses a *salt* at random. The salt can be application specific (e.g., a string identifying the application), a serial number within the application (e.g., the serial number of the message signed), or even a counter. However, a careful application would ensure that the salt contains enough randomness to be unpredicted.

In order to compute $HAIFA^C_{salt}(M)$ using the compression function $C : \{0,1\}^{m_c} \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^s \rightarrow \{0,1\}^{m_c}$ the message is first padded, and divided into l blocks of n bits each, $pad(M) = M_1||M_2||\ldots||M_l$. Now, the user:

1. Sets h_0 as the initial value (according to the procedure defined in Section 3.3).
2. Computes iteratively
$$h_i = C(h_{i-1}, M_i, \#bits, salt).$$
3. Truncates h_l (according to the procedure defined in Section 3.3).
4. Output the truncated value as $HAIFA^C_{salt}(M)$.

3.2 The Padding Scheme

Let n be the block length (e.g., $n = 512$ or 1024). The padding of a message M is:

1. Pad with a single bit of 1.
2. Pad with as many 0 bits as needed such that the length of the padded message (with the 1 bit and the 0's) is congruent modulo n to $(n - (t + r))$.
3. Pad with the message length encoded in t bits.
4. Pad with the digest length encoded in r bits.

We note that when a full padding block is added (i.e., the entire original message was already processed by the previous calls to the compression function, and the full message length was already used as an input to the previous call as the $\#bits$ parameter), the compression function is called with the $\#bits$ parameter set to **zero**. This property ensures that the additional full padding block is processed with a different $\#bits$ parameter than in prior invocations.

3.3 Variable Digest Length

Different digest lengths are needed for different applications. HAIFA supports variable digest length while preventing relations between the digests of the same message with different hash sizes. For generating a digest of length m,

1. The initial value h_0 is computed by $h_0 = C(MIV, m, 0, 0)$, where MIV is a master IV, and m is encoded as the content of the block.
2. The digest length is used by the padding schemes, and thus directly affects the compression of the last block.
3. After the final block is processed, the digest is composed of m bits of the last computed chaining value h_l.

Note that h_0 can be computed during the initialization of the hash function or can be computed in advance, and be hard-coded into the implementation.

3.4 The Security of HAIFA Hash Functions

The HAIFA mode of iteration preserves many useful properties of the compression function. If the compression function is collision resistant, so does the hash function. The same is true with respect to PRF features of the compression function, i.e., if the compression function is PRF, then so does the hash function. This makes HAIFA ideal for message authentication codes besides hashing. HAIFA also offers maximal security against (second) preimage attacks.

3.4.1 Collision Resistance Preservation The proof that HAIFA preserves the collision resistance of the compression function is very similar to the one used to prove that Merkle-Damgård hash functions retain the collision resistance of the underlying compression function.

As HAIFA uses salts, we shall consider the strongest definition of a collision in the compression function where the adversary may control all input parameters to the compression function and tries to generate the same output. We assume under this strong assumption that the adversary can even manipulate the $\#bits$ parameter.

Let M_1 and M_2 be the two colliding messages, i.e., $HAIFA^C_{salt_1}(M_1) = HAIFA^C_{salt_2}(M_2)$, with respective lengths l_1 and l_2. If the lengths l_1 and l_2 are different, or the salts are different, then the last blocks are necessarily different. Therefore, a collision of the (full) hash function allows to find a collision of the compression function of the last block.

If the lengths of the messages are the same and the salts are the same, one can start from the equal digest and equal last block and trace backwards till the point where the inputs to the compression function (either the input block or the input chaining value) differ, as at some point they must differ (otherwise $M_1 = M_2$). The same argument as the one for the Merkle-Damgård mode shows that there must exist a message block i such that $M^1_i \neq M^2_i$ or $h^1_{i-1} \neq h^2_{i-1}$ (where the superscript denotes the corresponding message), for which $C(h^1_{i-1}, M^1_i, salt, \#bits) = C(h^2_{i-1}, M^2_i, salt, \#bits)$, i.e., a collision of the compression function is found.

3.4.2 Security Against Extension Attacks HAIFA uses a bit counter, which is processed in each and every compression function call. This extra input offers great security advantages, one of which is the prefix-free encoding of the inputs to the compression function, which is independent of the messages themselves.

The reason for that is that the last block (or the one before it, in case an additional padding block is added) is compressed with the number of bits that were processed so far. If this value is not a multiple of the block size then the resulting chaining value is not equal to the chaining value that is needed to extend the message. If the message is a multiple of a block, then an additional block is hashed with the parameter $\#bits = 0$. Thus, the chaining value required for the extension remains obscure to the adversary.

We conclude that as long as the compression function is secure, it is not possible to compute $HAIFA_{salt}(m||x)$, given $HAIFA_{salt}(m)$ for any x (even if $salt$ is given). This has some interesting security features (besides the obvious suitability for simpler MAC constructions).

3.4.3 PRF Preservation and PRO Preservation The bit counter scheme allows a prefix-free encoding of the message. Among other things, this fact proves that HAIFA (when instantiated with a random oracle as a compression function) preserves the pseudorandom oracle property [19].

The prefix-free encoding also ensures the preservation of the pseudorandom function property of the compression function [6]. And thus, the only way to distinguish a HAIFA hash function effectively from a random string/random oracle is to use internal collisions, providing security of $\min\{2^m, 2^{m_c/2}\}$ against these attacks. The 2^m option is for cases where more than half of the bits of the internal state are truncated (and thus, "exhaustive search"-like attacks require less effort than attacks based on internal collisions).

3.4.4 Security Against Second Preimage Attacks HAIFA offers full security against second preimage attacks, i.e., finding a second preimage or a chosen target preimage of an m-bit digest requires 2^m compression functions calls. We first consider some of the latest results on Merkle-Damgård, and show that they do not apply to HAIFA. We then discuss some theoretical reasoning why even future attacks are expected to fail.

- **Dean's expandable message technique (second preimage attack)** — Dean's attack [25] is based on finding fix-points for the compression function, which can be iterated repeatedly. While it maybe easy to find a fix-point for an instance of the compression function, the use of the *#bits* counter prevents the repeated concatenation of the fixed-point to itself (as for different *#bits* different fix-points are expected). Moreover, even if a fixed-point for multiple *#bits* value is found, the phase of connecting the expandable message to the target message requires that the adversary commits to a specific location (i.e., which message block is replaced), which means, that the connection of the expandable message to the challenge message requires 2^{m_c} operations.
- **Kelsey and Schneier's expandable message (second preimage attack)** — Kelsey and Schneier's attack [37] is based on constructing an expandable message using Joux's multicollision technique [33]. As noted before, even if such a message was constructed, the cost of connecting it to the challenge message is like the cost of finding a second preimage of the compression function. Moreover, to generate the expandable message, one needs to be able to connect from a given chaining value two sequences of blocks with differing lengths resulting in a common chaining value that can be connected to different positions in the sequence of chaining values. The best possible approach would be to set the length after one block of this multicollision (e.g., block l), find a one block/two block collision that leads to the given length (i.e., start from position $l-1$ and find a message block that leads to a collision with a two message block starting at position $l-2$) and find from this location a collision between one message block with three blocks. The result is a limited "expandable message" of between two and five blocks which must be used starting at block $l-2$ (and then its length is either 3 or 5 blocks) or at block $l-1$ (and then its length is either 2 or 4 blocks). All other options are foiled by the *#bits* parameter which has to be determined in other locations as well.
- **The Herding Attack (chosen target preimage attacks)** — In the herding attack [36], the adversary constructs a diamond structure, a set of many chaining values from which the adversary knows how to get to a specific target value. As HAIFA contains salts, the adversary has either to choose the salt on his own (making the attack scenario less realistic) or generate a diamond structure for every possible salt. As the salt length is equal to the digest size (or even longer), this approach takes pre-processing time which is larger than a preimage attack and whose memory storage makes standard time-memory attacks more favorable. Moreover, unlike the Merkle-Damgård construction where the same diamond structure can be used in any possible location, in HAIFA, the diamond structure is fixed to a given location in the stream due to the *#bits* parameter, thus reducing the applicability of the attack.

– **Second Preimage Attack Based on Herding** — The latest second preimage attack suggested in [1] uses a diamond structure to allow the adversary to generate short "patches" to the message, and thus obtain a second preimage attack which is slightly slower than other techniques, but at the same time can deal with more hash function constructions. As stated earlier, the fact that the diamond structure is fixed to a given position, renders this impractical (or more precisely makes this attack equivalent to exhaustive search against the compression function).

While the above issues deal with concrete attacks, one might ponder whether there are other second preimage attacks which may break HAIFA hash functions. Though the general case is not yet solved, the results of [17] claim that if the compression function is a random oracle, then indeed there is no a shortcut second preimage attack on HAIFA.

The main reason for the security is that the bit counter prevents applying any attack in more than one specific location (i.e., the adversary has to commit in advance to the location where the second preimage is to be found) even in the theoretical settings studied in [17]. Hence, the best strategy an adversary could apply is to try and find a single block second preimage, which requires an exhaustive search if the compression function is strong.

3.4.5 The Security Advantages of the Salt The *salt* parameter can be considered as defining a family of hash functions as needed by the formal definitions of [48] in order to ensure the security of the family of hash functions. This parameter can also be viewed as an instance of the randomized hashing concept [29], thus, it also inherits all the advantages of the two concepts:

– The ability to define the security of the hash function in the theoretical model.
– Transformation of all attacks on the hash function that can use precomputation from an off-line part and an on-line part to only on-line part (as the exact *salt* is not known in advance).
– Increasing the security of digital signatures, as the signer chooses the *salt* value, and thus, any attack aiming at finding two messages with the same hash value has to take the *salt* into consideration. See [29] for more details about this property.

We note that the salt can be application specific (e.g., a string identifying the application), a serial number that follows the application (e.g., the serial number of the message signed), a counter, or a random string. It is obvious that the salt can also be set as a combination of these values. However, we emphasize that applications that need the extra security suggested by the salt should use as many random bits of salt as possible.

4 Specifications of SHAvite-3

SHAvite-3 has two flavors, according to the used compression function and digest size:

1. SHAvite-3_{256} uses the compression function C_{256} and produces digests of up to 256 bits,
2. SHAvite-3_{512} uses the compression function C_{512} and produces digests of 257 to 512 bits.

Specifically, digest lengths 160, 224, and 256 bits (required by the NIST call, as well as needed for a SHA-1 replacement) are to be produced by SHAvite-3_{256} (with truncation, as defined by HAIFA). The digest lengths 384 and 512 bits required by the call are to be produced by SHAvite-3_{512}.

4.1 Specifications of SHAvite-3_{256}

SHAvite-3_{256} is a HAIFA hash function, based on the compression function C_{256}. The compression function C_{256} accepts a chaining value of 256 bits (i.e., $m_c = 256$), a message block of size 512 bits ($n = 512$), a salt of size 256 bits ($s = 256$), and a bit counter of 64 bits ($b = 64$).

We use an underlying block cipher E^{256} in a Davies-Meyer transformation to construct C_{256}. The block cipher is a 12-round Feistel block cipher. Each round function of the block cipher is composed of three full rounds of AES. The plaintext size is 256 bits (the 256 bits of the chaining value), while the "key" (composed of the message block, the salt, and the counter) size is $512 + 64 + 256 = 832$ bits. Not all the "key" bits are treated equally, as 512 of these bits are the message block, 64 bits are the bit counter, and the remaining 256 bits are the salt.

4.1.1 The C_{256}'s Underlying Block Cipher — E^{256} The block cipher accepts a 256-bit plaintext P, treated as an array of eight 32-bit words $P[0, \ldots, 7]$. The plaintext is divided into two halves $P = (L_0, R_0)$, where L_0 contains words $P[0, \ldots, 3]$, and R_0 contains words $P[4, \ldots, 7]$. We note that bytes 0,1,2,3 of L_0 are $P[0]$, while bytes 12,13,14,15 of R_0 are $P[7]$. Then, the round function is repeated 12 times:

$$(L_{i+1}, R_{i+1}) = (R_i, L_i \oplus F^3_{RK_i}(R_i)).$$

$F^3(\cdot)$ accepts an input of 128 bits, R_i, as well as a 384-bit subkey, $RK_i = (k_i^0, k_i^1, k_i^2)$, and applies three full rounds of AES, using k_i^0 as a whitening key before the first internal round, k_i^1 the subkey of the first round, k_i^2 the subkey of the internal second round (and all zeroes as the subkey of the third internal round):

$$F^3_{(k_i^0,k_i^1,k_i^2)}(x) = AESRound_{0^{128}}(AESRound_{k_i^2}(AESRound_{k_i^1}(x \oplus k_i^0))).$$

We note that the last round's subkey (which is XORed) is the all zero value (thus, this operation can be omitted). We also note that all the AES rounds are full AES rounds (with the MixColumns operation).

The ciphertext $C = (L_{12}, R_{12})$ is the output of the block cipher, where bytes 0,1,2,3 of L_{12} compose the first 32-bit word of the ciphertext. We outline the block cipher E^{256} in Figure 2.

4.1.2 The Message Expansion The message expansion of C_{256} (the key schedule algorithm of E^{256}) accepts a 512-bit message block, a 64-bit counter, and a 256-bit salt. All are treated as arrays of 32-bit words (containing 16, 2, and 8 words, respectively), which are used to generate 36 subkeys of 128 bits each, or a total of 144 32-bit words.

Let $rk[0, \ldots, 143]$ be an array of 144 32-bit words, let $msg[0, \ldots, 15]$ be the message array (of 32-bit each), $cnt[0, 1]$ be the counter array (we parse the 64-bit counter $\#bits$ as a two word array, where $cnt[0]$ contains the least significant part of $\#bits$), and $salt[0, \ldots, 7]$ be the salt.

The first 16 words of $rk[\cdot]$ are initialized with the message words themselves. After that we repeat a process that generates 16 words in a nonlinear manner and then 16 words in a linear manner. The nonlinear process takes four $rk[\cdot]$ words, encrypts them under the salt (twice under the first four words of the salt, and twice under the last four words of the salt), and XORs the outcome with four (other) $rk[\cdot]$ words to produce the next four words (this is repeated four times in each iteration of the nonlinear process). The linear process takes two words from $rk[\cdot]$ and XORs them to produce the next word (this is repeated sixteen times in each iteration of the linear process).

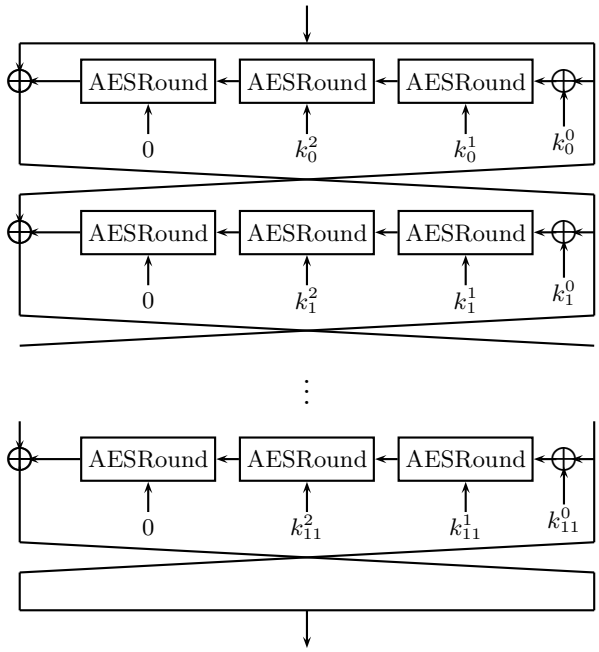


Fig. 2. The underlying block cipher of C_{256}

Eight of the produced words are XORed with the counter (four with $cnt[0]$ and four with $cnt[1]$), thus preventing any slide properties of the cipher: $rk[16]$, $rk[54]$, $rk[91]$, and $rk[124]$ are XORed with $cnt[0]$ during their update, and $rk[17]$, $rk[53]$, $rk[90]$, and $rk[127]$ are XORed with $cnt[1]$.

A summary of the computation of rk is as follows:

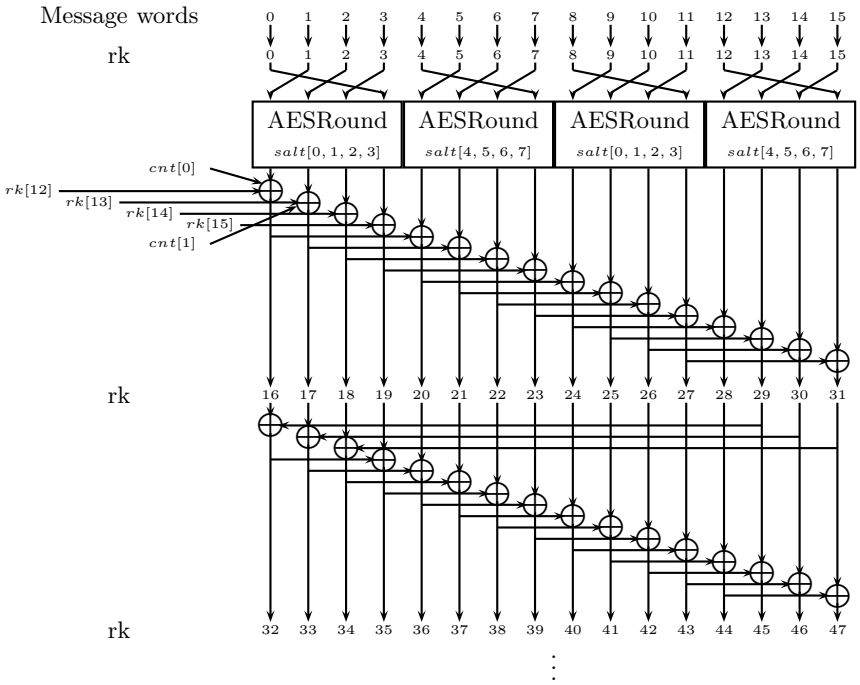
- For $i = 0, \ldots, 15$ set $rk[i] \leftarrow msg[i]$.
- Set $i \leftarrow 16$
- Repeat four times:
  1. **Nonlinear Expansion Step:** Repeat twice:
     (a) Let

     $$t[0..3] = AESRound_0((rk[i-15]||rk[i-14]||rk[i-13]||rk[i-16]) \oplus (salt[0]||salt[1]||salt[2]||salt[3])).$$

     (b) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i+j-4]$.
     (c) If $i = 16$ then $rk[16] \oplus = cnt[0]$ and $rk[17] \oplus = cnt[1]$.
     (d) If $i = 84$ then $rk[86] \oplus = cnt[1]$ and $rk[87] \oplus = cnt[0]$.
     (e) $i \leftarrow i + 4$.
     (f) Let

     $$t[0..3] = AESRound_0((rk[i-15]||rk[i-14]||rk[i-13]||rk[i-16]) \oplus (salt[4]||salt[5]||salt[6]||salt[7])).$$

     (g) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i+j-4]$.
     (h) If $i = 56$ then $rk[57] \oplus = cnt[1]$ and $rk[58] \oplus = cnt[0]$.
     (i) If $i = 124$ then $rk[124] \oplus = cnt[0]$ and $rk[127] \oplus = cnt[1]$.



The salts are XORed to the inputs before the SubBytes operations.
We note that the counters are added in different positions in different iterations of the nonlinear expansion step.

Fig. 3. The Message Expansion of C_{256}

(j) $i \leftarrow i + 4$.

2. **Linear Expansion Step:** Repeat sixteen times:
   (a) $rk[i] \leftarrow rk[i-16] \oplus rk[i-3]$.
   (b) $i \leftarrow i + 1$.

Figure 3 outlines the message expansion algorithm.

Once $rk[\cdot]$ is initialized, its 144 words are parsed as 36 subkeys of 128-bit each, which are then used as 12 triplets of subkeys, i.e.,

$$
\begin{aligned}
RK_0 &= (k_0^0, k_0^1, k_0^2) = ((rk[0], rk[1], rk[2], rk[3]),\ (rk[4], rk[5], rk[6], rk[7]),\ (rk[8], rk[9], rk[10], rk[11])) \\
RK_1 &= (k_1^0, k_1^1, k_1^2) = ((rk[12], rk[13], rk[14], rk[15]),\ (rk[16], rk[17], rk[18], rk[19]), \\
&\qquad (rk[20], rk[21], rk[22], rk[23])) \\
&\ \ \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
RK_i &= (k_i^0, k_i^1, k_i^2) = ((rk[12 \cdot i], rk[12 \cdot i + 1], rk[12 \cdot i + 2], rk[12 \cdot i + 3]), \\
&\qquad (rk[12 \cdot i + 4], rk[12 \cdot i + 5], rk[12 \cdot i + 6], rk[12 \cdot i + 7]), \\
&\qquad (rk[12 \cdot i + 8], rk[12 \cdot i + 9], rk[12 \cdot i + 10], rk[12 \cdot i + 11])) \\
&\ \ \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{aligned}
$$

4.1.3 Summary of C_{256} Each compression function call to C_{256} has four inputs. The message block M_i, the salt $salt$, and the bit counter $\#bits$ are viewed as a key of the block cipher E^{256}, while the chaining value h_{i-1} is treated as the plaintext. Then,

$$h_i = C_{256}(h_{i-1}, M_i, salt, \#bits) = E^{256}_{M_i||\#bits||salt}(h_{i-1}) \oplus h_{i-1}.$$

4.1.4 Generating Digests of up to 256 Bits In order to hash the message M into an m-bit digest, for $m \leq 256$, first compute IV_m which is

$$h_0 = IV_m = C_{256}(MIV_{256}, m, 0, 0),$$

where

$$MIV_{256} = C_{256}(0,0,0,0) = \texttt{62F4F5C5 DFDE7372 B1363F69 D90121BA}$$
$$\texttt{0DB1E936 56E69E08 9DBD1479 DE6E4E0F}_x.$$

Table 2 lists the values of IV_m for 224, 256, and 160-bit versions of SHAvite-3_{256}.

Let $|M|$ be the length of M before padding, measured in bits. Pad the message M according to the padding scheme of HAIFA:

1. Pad a single bit of 1.
2. Pad as many 0 bits as needed such that the length of the padded message (with the 1 bit and the 0's) is congruent modulo 512 to 432.
3. Pad $|M|$ encoded in 64 bits.
4. Pad m encoded in 16 bits.

Now, divide the padded message $pad(M)$ into 512-bit blocks, $pad(M) = M_1||M_2||\ldots||M_l$, and perform:

1. Set $\#bits \leftarrow 0$.
2. Set $h_0 \leftarrow IV_m$.
3. For $i = 1, \ldots, \lfloor |M|/512 \rfloor$:
   - Set $\#bits \leftarrow \#bits + 512$.
   - Compute $h_i = C_{256}(h_{i-1}, M_i, \#bits, salt)$.
4. - If the message length is a multiple of the block length ($|M| = 0 \bmod 512$), compute $h_l = C_{256}(h_{l-1}, M_l, 0, salt)$ (where M_l is a full padding block), else
   - If the message length allows for the padding to be in the same block as the last message block (i.e., $|M| \bmod 512 \leq 431$), compute $h_l = C_{256}(h_{l-1}, M_l, |M|, salt)$, else
   - Process some of the padding block with the last block containing message, i.e., compute $h_{l-1} = C_{256}(h_{l-2}, M_{l-1}, |M|, salt)$, and then compute $h_l = C_{256}(h_{l-1}, M_l, 0, salt)$ for processing the additional (partial) padding block.
5. Output $truncate_m(h_l)$, where $truncate_m(x)$ outputs the m leftmost bits of x, i.e., $x[0]||x[1]|\ldots$.

4.2 Specifications of SHAvite-3_{512}

SHAvite-3_{512} is a HAIFA hash function, based on the compression function C_{512}. The compression function C_{512} accepts a chaining value of 512 bits (i.e., $m_c = 512$), a message block of size 1024 bits ($n = 1024$), a salt of size 512 bits ($s = 512$), and a bit counter of 128 bits ($b = 128$).

IV_m	Value ($IV_m[0]\|\|IV_m[1]\|\|\ldots IV_m[7]$)			
IV_{160}	63128A01	3047C73E	83B982ED	E6F9DAE2
	375B2554	F79A82F6	E7D69EB1	A3698BC4$_x$
IV_{224}	D617833B	68EA6C8F	FF3DF700	E5B807EF
	6FDB4E75	F966482E	3B40F9B2	755891B2$_x$
IV_{256}	AE9F3281	5F867848	0C988766	D00B409D
	31C1F23F	5361AAB9	FB5E1BF6	889EE275$_x$

Table 2. IV_m for Common Values of the Digest Size

C_{512} is constructed similarly to C_{256}, as a Davies-Meyer transformation of a block cipher. The underlying block cipher has 14 rounds, and has a generalized Feistel structure. The plaintext size is 512 bits (the 512 bits of the chaining value), while the "key" (composed of the message block, the salt, and the counter) size is $1024+128+512 = 1664$ bits. The plaintext (the chaining value) is divided into four 128-bit words, and each round two of these 128-bit words enter the nonlinear round function and affect the other two (each word enters one nonlinear function and affect one word). The round function is composed of four full rounds of AES.

4.2.1 The C_{512}'s Underlying Block Cipher — E^{512} The block cipher accepts a 512-bit plaintext P, treated as an array of sixteen 32-bit words $P[0,\ldots,15]$. The plaintext is divided into four 128-bit words $P = (L_0, A_0, B_0, R_0)$, where L_0 contains words $P[0,\ldots,3]$, and R_0 contains words $P[12,\ldots,15]$. We note that bytes 0,1,2,3 of L_0 are $P[0]$, while bytes 12,13,14,15 of R_0 compose $P[15]$. Then, the round function is repeated 14 times:

$$(L_{i+1}, A_{i+1}, B_{i+1}, R_{i+1}) = (R_i, L_i \oplus F^4_{RK_{0,i}}(A_i), A_i, B_i \oplus F^4_{RK_{1,i}}(R_i)).$$

$F^4(\cdot)$ accepts an input of 128 bits R_i as well as 512-bit subkey $RK_{i,j} = (k^0_{i,j}, k^1_{i,j}, k^2_{i,j}, k^3_{i,j})$, and applies four rounds of AES, using k^0_i as a whitening key before the first internal round, and where the AddRoundKey operation of the fourth internal round is omitted (or done with the all-zero key).

$$F^4_{(k^0_{i,j},k^1_{i,j},k^2_{i,j},k^3_{i,j})}(x) = AESRound_{0^{128}}(AESRound_{k^3_i}(AESRound_{k^2_i}(AESRound_{k^1_i}(x \oplus k^0_i)))).$$

As in E^{256}, the last round's subkey is the all-zero key, and all the rounds are full AES rounds (i.e., with the MixColumns operation).

The ciphertext $C = (L_{14}, A_{14}, B_{14}, R_{14})$ is the output of the block cipher, where bytes 0,1,2,3 of L_{14} compose the first 32-bit word of the ciphertext. We outline the block cipher E^{512} in Figure 4.

4.2.2 The Message Expansion of C_{512} The message expansion of C_{512} (the key schedule algorithm of E^{512}) accepts a 1024-bit message block, 128-bit counter, and 512-bit salt. All are treated as arrays of 32-bit words (of 32, 4, and 16 words, respectively), which are used to generate 112 subkeys of 128 bits each, or a total of 448 32-bit words.

Let $rk[\cdot]$ be an array of 448 32-bit words, let $msg[0,\ldots,31]$ be the message array, $cnt[0,\ldots,3]$ be the counter array, and $salt[0,\ldots,15]$ be the salt. The first 32 words of rk are initialized with the message words themselves. Then, we repeat a process that generates 32 words in a nonlinear manner and 32 words in a linear manner. Sixteen of the produced words are XORed with the counter (four with each $cnt[i]$). The computation of $rk[\cdot]$ is done as follows:

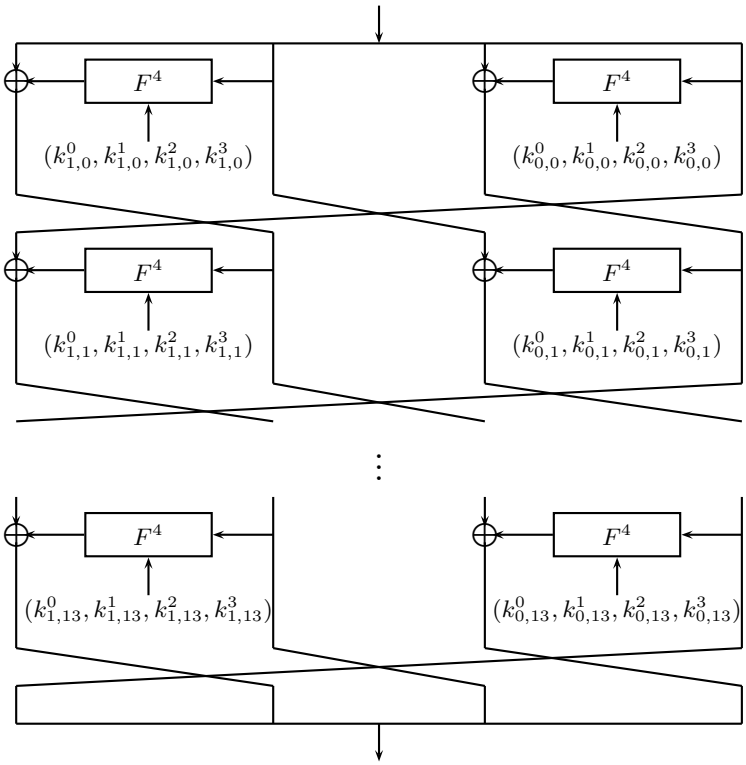


Fig. 4. The Underlying Block Cipher of C_{512}

– For $i = 0, \ldots, 31$ set $rk[i] \leftarrow msg[i]$.
– Set $i \leftarrow 32$
– Repeat six times:
  1. **Nonlinear Expansion Step:** Repeat twice:
     (a) Let

$$t[0..3] = AESRound_0((rk[i-31]||rk[i-30]||rk[i-29]||rk[i-32]) \oplus (salt[0]||salt[1]||salt[2]||salt[3])).$$

     (b) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
     (c) If $i = 32$ then $rk[32] \oplus = cnt[0]$, $rk[33] \oplus = cnt[1]$, $rk[34] \oplus = cnt[2]$, and $rk[35] \oplus = cnt[3]$.
     (d) $i \leftarrow i + 4$.
     (e) Let

$$t[0..3] = AESRound_0((rk[i-31]||rk[i-30]||rk[i-29]||rk[i-32]) \oplus (salt[4]||salt[5]||salt[6]||salt[7])).$$

     (f) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
     (g) If $i = 164$ then $rk[164] \oplus = cnt[3]$, $rk[165] \oplus = cnt[2]$, $rk[166] \oplus = cnt[1]$, and $rk[167] \oplus = cnt[0]$.
     (h) $i \leftarrow i + 4$.

(i) Let

$$t[0..3] = AESRound_0((rk[i-31]||rk[i-30]||rk[i-29]||rk[i-32]) \oplus (salt[8]||salt[9]||salt[10]||salt[11])).$$

(j) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
(k) If $i = 440$ then $rk[440] \oplus = cnt[1]$, $rk[441] \oplus = cnt[0]$, $rk[442] \oplus = cnt[3]$, and $rk[443] \oplus = cnt[2]$.
(l) $i \leftarrow i + 4$.
(m) Let

$$t[0..3] = AESRound_{0^{128}}((rk[i-31]||rk[i-30]||rk[i-29]||rk[i-32]) \oplus (salt[12]||salt[13]||salt[14]||salt[15])).$$

(n) For $j = 0, \ldots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
(o) If $i = 316$ then $rk[316] \oplus = cnt[2]$, $rk[317] \oplus = cnt[3]$, $rk[318] \oplus = cnt[0]$, and $rk[319] \oplus = cnt[1]$.
(p) $i \leftarrow i + 4$.

2. **Linear Expansion Step:** Repeat 32 times:
   (a) $rk[i] \leftarrow rk[i-32] \oplus rk[i-7]$.
   (b) $i \leftarrow i + 1$.

– Repeat the **Nonlinear Expansion Step** an additional time.

Once $rk[\cdot]$ is initialized, its 448 words are parsed as 112 words of 128-bit each, which are the subkeys (14 double quartets of 128-bit words each), i.e.,

$$\begin{aligned}
RK_{0,0} = (k^0_{0,0}, k^1_{0,0}, k^2_{0,0}, k^3_{0,0}) &= ((rk[0], rk[1], rk[2], rk[3]),\ (rk[4], rk[5], rk[6], rk[7]),\\
&\quad (rk[8], rk[9], rk[10], rk[11])\ (rk[12], rk[13], rk[14], rk[15]))\\
RK_{1,0} = (k^0_{1,0}, k^1_{1,0}, k^2_{1,0}, k^3_{1,0}) &= ((rk[16], rk[17], rk[18], rk[19]), (rk[20], rk[21], rk[22], rk[23])\\
&\quad ((rk[24], rk[25], rk[26], rk[27]), (rk[28], rk[29], rk[30], rk[31]))\\
\vdots \qquad\qquad \vdots &\qquad\qquad\qquad\qquad \vdots\\
RK_{0,i} = (k^0_{0,i}, k^1_{0,i}, k^2_{0,i}, k^3_{0,i}) &= ((rk[32\cdot i], rk[32\cdot i+1], rk[32\cdot i+2], rk[32\cdot i+3]),\\
&\quad (rk[32\cdot i+4], rk[32\cdot i+5], rk[32\cdot i+6], rk[32\cdot i+7]),\\
&\quad (rk[32\cdot i+8], rk[32\cdot i+9], rk[32\cdot i+10], rk[32\cdot i+11]),\\
&\quad (rk[32\cdot i+12], rk[32\cdot i+13], rk[32\cdot i+14], rk[32\cdot i+15]))\\
RK_{1,i} = (k^0_{1,i}, k^1_{1,i}, k^2_{1,i}, k^3_{1,i}) &= ((rk[32\cdot i+16], rk[32\cdot i+17], rk[32\cdot i+18], rk[32\cdot i+19]),\\
&\quad (rk[32\cdot i+20], rk[32\cdot i+21], rk[32\cdot i+22], rk[32\cdot i+23]),\\
&\quad (rk[32\cdot i+24], rk[32\cdot i+25], rk[32\cdot i+26], rk[32\cdot i+27]),\\
&\quad (rk[32\cdot i+28], rk[32\cdot i+29], rk[32\cdot i+30], rk[32\cdot i+31]))\\
\vdots \qquad\qquad \vdots &\qquad\qquad\qquad\qquad \vdots
\end{aligned}$$

4.2.3 Summary of C_{512} Each compression function call to C_{512} has four inputs. The message block M_i, the salt $salt$, and the bit counter $\#bits$ are viewed as a key of the block cipher E^{512}, while the chaining value h_{i-1} is treated as the plaintext. Then, the output of the compression function is

$$h_i = C_{512}(h_{i-1}, M_i, salt, \#bits) = h_{i-1} \oplus E^{512}_{M_i||\#bits||salt}(h_{i-1}).$$

IV_m	Value ($IV_m[0]\|\|IV_m[1]\|\|\ldots IV_m[15]$)			
IV_{384}	1E41CEC0	E742F23B	5E195589	DDCFE7A0
	827678F1	97AB48F6	5306C06C	00064879
	15FE61A9	79FFC139	10426AA1	F255945e
	5573B567	B9BDA1CA	CEF5447F	$1A4A03A7_x$
IV_{512}	8A671C48	21FBB075	6C11F5A0	2B153831
	C6192444	1254BA09	ADBD2BF9	6956353E
	51ECE04E	B38D02EC	3CCCC57B	B76EA6DA
	DDED39A5	ACB431B4	9452E478	$F2DCEE8D_x$

Table 3. IV_m for Common Values of the Digest Size

4.2.4 Generating Digests of 257 to 512 Bits In order to hash the message M into an m-bit digest, for $256 < m \leq 512$, first compute IV_m which is

$$h_0 = IV_m = C_{512}(MIV_{512}, m, 0, 0).$$

where

$$MIV_{512} = C_{512}(0,0,0,0) = \begin{matrix} \texttt{1FA9BAD2} & \texttt{9AD8E2A5} & \texttt{713898D1} & \texttt{7B528545} \\ \texttt{908EA84D} & \texttt{035E2C9E} & \texttt{57C1E9A0} & \texttt{74392F7F} \\ \texttt{F0D780C2} & \texttt{298519CD} & \texttt{E387BCEC} & \texttt{12261052} \\ \texttt{EEB5CE28} & \texttt{A005D17A} & \texttt{6558949A} & \texttt{EFAFDDF1}_x. \end{matrix}$$

As before, let $|M|$ be the length of the message M before padding, measured in bits. Pad the message M according to the padding scheme of HAIFA:

1. Pad a single bit of 1.
2. Pad as many 0 bits as needed such that the length of the padded message (with the 1 bit and the 0's) is congruent modulo 1024 to 880.
3. Pad $|M|$ encoded in 128 bits.
4. Pad m encoded in 16 bits.

Now, divide the padded message $pad(M)$ into 1024-bit blocks, $pad(M) = M_1||M_2||\ldots||M_l$, and perform:

1. Set $\#bits \leftarrow 0$.
2. Set $h_0 \leftarrow IV_m$.
3. For $i = 1, \ldots, \lfloor |M|/1024 \rfloor$:
   - Set $\#bits \leftarrow \#bits + 1024$.
   - Compute $h_i = C_{512}(h_{i-1}, M_i, \#bits, salt)$.
4. - If $|M| = 0 \bmod 1024$, compute $h_l = C_{512}(h_{l-1}, M_l, 0, salt)$, else
   - If $|M| \bmod 1024 \leq 879$, compute $h_l = C_{512}(h_{l-1}, M_l, |M|, salt)$, else
   - Compute $h_{l-1} = C_{512}(h_{l-2}, M_{l-1}, |M|, salt)$, and compute $h_l = C_{512}(h_{l-1}, M_l, 0, salt)$.
5. Output $truncate_m(h_l)$.

Table 3 lists the values of IV_m for 384-bit and 512-bit digests produced by SHAvite-3_{512}.

4.3 Degenerate Salts

In applications where a salt cannot be used or when the additional functionality is not needed (possibly in exchange for loss in security) it is possible to use a fixed salt. While better security can be achieved if any such application would have its own salt, in certain cases an agreed fixed salt would better be used. As all fixed salts presumably have the same strength, we suggest the use of the all-zero salt in these cases. This salt can be hardcoded into unsalted implementations. The speed using such hardcoded salt (in particular the all-zero salt) is expected to be slightly faster than for the general case).
We note that the KAT/MCT answers for the NIST call were produced using these fixed salts.

5 Design Criteria and Rationale

In the recent few years, several advances in hash functions cryptanalysis were reported. These results show that small nonlinearity in bit-wise operations, spread using a mixture of XORs, modular additions, and rotations, are insufficient to offer good security. Hence, in order to generate a good compression function, one has to use strong nonlinear components.

5.1 Designing the Compression Function

As we use the well-understood Davies-Meyer transformation, the problem of devising a secure compression function is reduced to the problem of constructing a secure block cipher. We have chosen a Feistel construction (or a generalized Feistel one) as a well understood construction, whose security properties are known. Adding to that the use of the AES round function, we obtain a secure compression function.

We use consecutive rounds of AES in the round function, and adapt the results on the security of AES to the security of SHAvite-3. This decision also allows the adaption of optimization techniques used for AES, offering an efficient construction. This also ensures that devices which need to include both AES and SHAvite-3 are expected to do so with fewer gates for both of them than in the case of AES and SHA-512 (for example). The same holds for software packages which include the two primitives. Thus, our choice makes the development and certification of products easier, as the AES round implementation can be done and verified only once. This fact is expected to shorten the time required for the deployment of SHAvite-3.

We have set several requirements for the security of the compression functions in use: First of all, any differential characteristic of the block cipher should have a very low probability (i.e., less than 2^{-m}). This is the first step in offering a secure cipher, but as presented in the recent results, this is not sufficient, as the adversary can control the key (message). This extra control allows the adversary to find right pairs with respect to the differential despite its low probability, by picking messages that lead to satisfaction of some differential transitions. Thus, we aim at reducing the probability of any related-key differential of the block cipher, where the adversary has control over the key as well. We therefore ensure that the best related-key differential would has as low differential probability as possible (it may be at most $2^{-m/2}$ for m-bit digest to prevent any collision attacks, but need to be about 2^{-m} to prevent any differential-based second preimage attacks).

In order to ensure that the differential properties of the key schedule would not interact in an unpredictable manner with the "encryption" part of the block cipher, we designed the message expansion with two types of operations — AES based operations which offers high nonlinearity, and linear operations which offer diffusion and "break" the sequence of AES operations.

5.2 Designing the Mode of Iteration

For the iteration method, we considered the following modes of iteration:

- Merkle-Damgård — Following the recent results on the lack of second preimage resistance, we find the use of this mode unfit for a modern hash function.
- Tree-hash — While modes of iterations based on trees offer a great deal of parallelization in the implementation, they suffer from various flaws. For example, a preimage attack faster than exhaustive search on tree hash is presented in [1]. This lack of security, as well as the fact that the memory requirements for a tree hash functions are too large for constrained environments (such as smart cards), make tree hashes unsuitable for SHA-3.
- Sponge constructions — Even though sponge constructions have strong theoretical foundations [10], we identify two issues concerning their use. The first is the fact that the internal state is large, making it unsuitable for constrained environments, and may lead to performance penalties when sufficient cache memory is not available. The second issue concerns the gap between theory and practice. A sponge construction is secure if the round function is strong as a whole. However, as the internal state is large, such a function is expected to be very slow and hard to analyze. Explicit constructions solve this issue by using a weak round function, a very dangerous practice [21, 46].
- Widepipe — This mode offers security with a relatively small performance penalty [39]. Double internal state (i.e., $m_c \geq 2m$) is the minimal size that suggests full (second) preimage resistance of 2^m for an m-bit digest. It requires the function to handle twice the number of bits of the chaining value. Even this expansion rate of two causes a loss of resources (more gates are needed to store the chaining value, the compression function has to process more bits, etc.) and thus undesirable if can be avoided.
- HAIFA — HAIFA offers security against all known cryptanalytic attacks on modes of iteration, while incurring no (or very little) performance penalties.

As can easily be seen, HAIFA is the best solution that suggests full security without increasing the internal state. The performance penalty of a larger state in a hardware implementation is apparent. At a first glance, the penalty in software may be considerably small. However, when multiple instances of the hash function are being run in parallel or when the available cache memory is small, each additional cache-miss increases the running time of the hash function significantly.

The choice of HAIFA is thus natural. It offers the best performance for the required security for an m-bit hash function with collision resistance of $2^{m/2}$ and (second) preimage resistance of 2^m.

5.3 Choices of Parameters

The choice of parameters was motivated by several arguments. First, we decided to maintain the same input parameters as the SHA-2 family. This decision was made to facilitate an easy transition for any application that is expected to use SHAvite-3, allowing for a faster development and deployment.

The salt must be at least half the size of the chaining value in order to protect against herding attacks (i.e., $s > m_c/2$). Nevertheless, we decided to pick the salt size equal to the chaining value size, for the sake of applications that use SHAvite-3 in ways where an m-bit digest requires a 2^m security, which we wanted to maintain even against attacks targeting the salt.

5.4 Choices of Constants

SHAvite-3 uses a relatively small number of constants. There are constants used in the AES round over which we have no control. The only constants of SHAvite-3 we chose are the values of MIV_{256}, MIV_{512}, the "tap" positions in the message expansion, and the locations of where the counter and salt are mixed.

In order to allow implementations to save memory, we decided to pick $MIV_{256} = C_{256}(0,0,0,0)$ and $MIV_{512} = C_{512}(0,0,0,0)$. These values can easily be computed on the fly, or precomputed and stored. In any case, we just tried to pick relatively random strings (as much as a fixed string can be random), without resorting to the standard set of "common constants" (i.e., $\sqrt{2}, \varphi, \ldots$).

As for the tap positions, we first note that we use a register of 16 32-bit words in SHAvite-3_{256} and 32 words in SHAvite-3_{512} in the message expansion. We choose to perform a layer of AES rounds and only then to perform the XORs (of the nonlinear expansion step), in order to allow parallel computation of all the AES rounds of nonlinear expansion step.

In SHAvite-3_{256}, we have chosen the feedback taps of the linear feedback register to be 3 and 16 (i.e., $rk[i] = rk[i-3] \oplus rk[i-16]$). This choice ensures that the entire process is reversible and offers a good diffusion. Similarly, the choice of 7 and 32 ensures it in SHAvite-3_{512}.

For the locations where the counter is mixed, we chose locations which allow good mixing of the counter, and prevent slide properties. The locations were chosen to be in the four different nonlinear expansion steps. We also chose that inside the nonlinear expansion step, the counter is mixed in different respective locations, and we ensured that the distance between the different locations is not regular, and does not repeat, thus foiling any slide property. In SHAvite-3_{512} we picked four out of the seven nonlinear update steps (taking the odd ones). Using similar considerations, we made sure to pick different respective locations in the nonlinear expansion step. Mixing the order of the counter words used each time only serves to further prevent any slide attack and prevent the existence of "weak" bit counters.

Hence, there are no slide properties in the underlying block ciphers. Even if (somehow) the attack generates the same state in different positions, the counters break this property. As the bit counters change between different invocations of the compression function, this also assures that such relations cannot exist for more than a short number of rounds even between different invocations of the compression function.

In order to reduce the memory consumption, and prevent the need of storing tables of constants (which costs memory and/or gate area), we decided that SHAvite-3 would not use any other round constants. Even though this may seem to weaken the hash function, there are two good argument why this is not the case. The first is the fact that some constants are embedded into the AES round constant, and we see no reason to add more (as the issue of weak inputs is solved by AES' constants and design criteria). The second reason is the fact that the only conceivable problem following the use of the same constants (of the AES round function) over and over again is the existence of slide properties. As noted before, this is impossible, and thus, we conclude that there is no need for additional round constants.

6 The Security of SHAvite-3

The security of SHAvite-3 is based on the security of its compression functions C_{256} and C_{512}, and the security of the mode of iteration used (HAIFA). Thus, we perform security analysis of each of these two parts independently, starting from the security of the compression functions.

6.1 The Security of the Compression Functions

SHAvite-3's compression functions are based on AES. We therefore recall a few results concerning the security of AES.

Lemma 1. ([35]) *The exact 2-round AES maximal expected differential probability is equal to* $53/2^{34} \approx 1.656 \cdot 2^{-29}$.

Lemma 2. ([35]) *The 4-round AES maximal expected differential probability is upper bounded by* $(53/2^{34})^4 \approx 1.881 \cdot 2^{-114}$.

The upper bound given in Lemma 2 is not tight. Under the assumption that 4-round AES behaves like a random permutation, the maximal expected differential probability is about $80 \cdot 2^{-128} = 2^{-121.7}$. This probability is derived from the Poisson distribution of the difference distribution table, whose maximal entry (besides the $0 \rightarrow 0$ entry) is expected to be $78 \cdot 2^{-128}$ or $80 \cdot 2^{-128}$ with an overwhelming probability [45].

As C_{256} uses 3-round AES as the round function, we conclude that:

Lemma 3. *The maximal expected differential probability of 3-round AES is upper bounded by* 2^{-49}.

Proof. Any 3-round differential can be decomposed into two (overlapping) 2-round differentials (one without the first round, and one without the last round). In each of these 2-round constructions, the differential with the highest probability has only one active AES' Super-box (2-round AES can be decomposed into 4 independent ciphers, each with input and output of 32 bits, called a Super-box). Having two active Super-boxes in the first (or the last) two rounds necessarily bound the probability of the differential to be no more than $(53 \cdot 2^{-34})^2 = 2809 \cdot 2^{-68} \approx 1.372 \cdot 2^{-57}$. Thus, a better differential (with higher probability) can have only one active Super-box in the first two rounds and in the last two rounds. This is possible if, and only if, there is one active S-box in the second round (being the last round of the first Super-box, and the first round of the second Super-box).

This differential has one active Super-box in the first two rounds, and four active S-boxes in the third round. Consider a possible output difference of this differential, and consider all the one-round characteristics which lead to this output difference. In each active S-box there are 127 input differences which lead to the desired output difference with non-zero probability. Now, consider the second round. If it has only one active S-box, then there are only 255 possible differences after the second round, whose differences in the active bytes is linearly dependent (due to the MixColumns operation). In other words, setting the input difference to one of the active S-boxes of the third round, immediately fixes the input difference of the other active S-boxes. Hence, for the given output difference there are at most 127 possibilities for the first 2-round differentials which may lead to the desired output difference. In the computation of the upper bound, we take into consideration the fact that any 2-round differential has probability at most $53 \cdot 2^{-34}$, and that in the one-round characteristics there are four active S-boxes. For the given output difference in each active S-box there are 126 input differences with probability 2^{-7} and one with probability 2^{-6}, each of the one-round characteristics has probabilities between 2^{-24} and 2^{-28}. But not all the characteristics have probability 2^{-24}, as for each active S-box there is only one input difference with probability 2^{-6}. Therefore, we conclude that the maximal expected differential probability is upper bounded by

$$53 \cdot 2^{-34} \cdot \left(2^{-24} + 126 \cdot 2^{-28}\right) = 53 \cdot 142 \cdot 2^{-34-28} = 7526 \cdot 2^{-62} \approx 1.837 \cdot 2^{-50}.$$

□

Considering the above lemma, we follow to prove the following results concerning E^{256}:

Lemma 4. *The differential properties of E^{256}:*
Except for the trivial $0 \rightarrow 0$ characteristic,

- *There is no iterative differential characteristic of 2-round E^{256}.*
- *Any 4-round iterative differential characteristic E^{256} has probability lower than 2^{-147}.*
- *Any 3-round differential characteristic of E^{256} has probability of no more than 2^{-98}.*
- *Any 9-round differential characteristic of E^{256} has probability of no more than 2^{-294}.*

Proof.

- Any 2-round iterative differential characteristic of a Feistel cipher requires that both rounds have a zero output difference. As the round function of E^{256} is invertible, it follows that the input differences are zero as well which results in the trivial $0 \rightarrow 0$ characteristic.
- 4-round iterative characteristic cannot have two rounds with zero input difference (these two rounds cannot be next to each other, as this causes two rounds to have zero input difference, i.e., the whole difference is zero, and if they are separated by a non-zero input/output round, we obtain the same case as the 2-round iterative characteristics). Thus, the characteristic has at most one round with a zero input/output difference, i.e., has probability of at most $(2^{-49})^3 = 2^{-147}$.
- It can be easily seen that at least two rounds in any non-trivial 3-round characteristics have non-zero input difference. Therefore, the maximal probability is $(2^{-49})^2 = 2^{-98}$.
- Following the previous lemma, it is easy to see that any 9-round differential characteristic of C_{256} has at most probability of $(2^{-98})^3 = 2^{-294}$.

□

Lemma 5. *The differential properties of E^{512}:*
Except for the trivial $0 \rightarrow 0$ differential:

- *There is no iterative differential characteristic of 2-round E^{512}.*
- *Any 3-round differential characteristic of E^{512} has probability of no more than 2^{-226}.*
- *Any 9-round differential characteristic of E^{512} has probability of no more than 2^{-678}.*

Proof.

- As E^{512} can be represented as a Feistel block cipher with a bijective round function of 256 bits,[1] Hence, similarly to the case of E^{256}, there is no 2-round iterative differential characteristic.
- Recall that $F^4(\cdot)$, is composed of 4 AES rounds. Thus, the maximal expected differential probability of any non-zero differential of $F^4(\cdot)$ is less than 2^{-113}. Hence, if we look at the Feistel representation of E^{512} in each active round, the maximal expected differential probability is 2^{-113} (corresponding to only one of the $F^4(\cdot)$ being active). Any non-trivial differential cannot have two consecutive rounds with input difference zero. If there were two such non-consecutive rounds (out of the three) with input difference zero, then that would mean that the non-zero difference which entered the round function produced a zero output difference, which is impossible due to the bijectiveness of the round function. Hence, there are at least two active rounds, and the maximal differential probability of 3-round differential characteristics is at most $(2^{-113})^2 = 2^{-226}$.

[1] We note that in this representation there is also a re-ordering bit permutations before the first round and after the last round, similar to the initial permutation and final permutation of DES.

– Following the previous lemma, it is easy to see that any 9-round differential characteristic of E^{512} has a maximal expected differential probability of $(2^{-226})^3 = 2^{-678}$.

□

6.1.1 The Security of the Underlying Block Ciphers Following the previous lemmas, it is easy to see that the underlying block ciphers E^{256} and E^{512} offer security against differential cryptanalysis. While we did not discuss linear cryptanalysis, it is possible to offer similar assurances against linear cryptanalysis (even though its usage in the hash function context is unclear).

The block ciphers in use are also secure against other cryptanalytic attacks. For example, the low probability of the best non-trivial differentials even for a small number of rounds, suggest that boomerang attacks (or amplified boomerang attacks) are not applicable to the block ciphers, and indicate that the amplified boomerang attack in the context of hash functions is likely avoided [34].

As the underlying block ciphers E^{256} and E^{512} are not used as block ciphers, there seems to be no apparent reason to analyze their security against other cryptanalytic techniques. Still, for completeness, we present results concerning the security of the two block ciphers, showing that there security is indeed intact:

– **Linear Cryptanalysis** — Results similar to the differential results, can be obtained for linear hulls of 3-round AES and 4-round AES. In [35] the 2-round maximal expected linear probability is found to be $1.638 \cdot 2^{-28}$ and the 4-round maximal expected linear probability is upper bounded by $1.802 \cdot 2^{-110}$. Using a similar procedure as in the differential case, we obtain that the maximal expected linear probability of 3-round AES is no more than $2^{-47.4}$. Hence, linear cryptanalysis is likely to fail for both E^{256} and E^{512}.
– **Impossible Differential Cryptanalysis** — E^{256} is a Feistel block cipher with a bijective round function. This implies the existence of a 5-round impossible differential. However, due to the strong diffusion of the round function, we do not expect a longer impossible differential in E^{256}. For E^{512}, there is a 9-round impossible differential (a structural impossible differential suggested in [38]). This (along with the strong round function) suggest that it is secure against impossible differential as well.
– **Differential-Linear Cryptanalysis** — Given the probabilities of the best differentials and the best linear approximations, it is easy to see that there is no high probability differential-linear approximation in any of the underlying block ciphers.
– **Algebraic Approaches** — While the level of the threat algebraic attacks pose to block ciphers (and specifically to hash functions) is still open, we analyze the security of the underlying block cipher to this kind of attacks. Consider the round functions $F^3(\cdot)$ and $F^4(\cdot)$. As they are composed of AES rounds, the best possible algebraic relations are of quadratic nature over $GF(2^8)$. As each additional round doubles the degree, the expected degree of algebraic relations concerning the input and output of the round functions over $GF(2^8)$ is 8 in $F^3(\cdot)$ and 16 in $F^4(\cdot)$. The repetition of the rounds increases the algebraic degree very quickly, and is expected to reach the maximal value after a few rounds. Specifically, in the case of E^{256}, after four rounds, the expected degree of any relation is 2^9 (more than the actual possible degree), which means that it achieved the maximal possible degree. In the case of E^{512}, after four rounds, the expected degree of any relation is 2^{12}, which means that the maximal degree is achieved. Besides the high degree, it appears that the resulting equations are more dense than in AES, thus making algebraic attacks which exploit the sparse nature of the equations less likely to be applicable.

- **Slide Attacks** — Slide attacks exploit the self-similarity of the cipher. The standard solution to the problem is to use different round constants, but these are not found in SHAvite-3. Despite that, SHAvite-3's underlying block ciphers are secure against slide attacks due to the bit counters. The bit counters are added in positions which break any self similarity property that may exist. The only problematic case is when $\#bits = 0$, which happens only during initializations (where the adversary has no control over the inputs), and during the processing of a full padding block (again, where the adversary has no control over the inputs).
- **Square Attacks** — SHAvite-3 uses the AES building block which is susceptible to square attacks in small number of rounds. The longest square properties that can be found are of four consecutive AES rounds, and using the Feistel structure of the underlying block ciphers, we can use them to find square properties of up to three rounds of E^{256} and E^{512} at most. Hence, the underlying block ciphers are secure against Square attacks.

6.1.2 Resistance to Collision Attacks on the Compression Function Without loss of generality we discuss the case of C_{256} — the results in the case of C_{512} are much stronger.

We will now assume that a random salt has been selected, and we will approximate the probability of a differential given that fixed random salt.

Clearly, due to the properties of an AES round, at least 5 bytes are active in the input and output in any layer of AES rounds in the message schedule of SHAvite-3. Without loss of generality, we concentrate on the second AES layer. We can assume that the differences in these 5 bytes are distinct (and even linearly independent) - the assumption that the salt is selected at random assures that the attacker cannot select the exact differences to his favorable values. The linear transforms evolve each of the (at least) 5 independent differences into at least 4 S-boxes at the input of the next AES layer (third layer) (or at the output of the first layer, in case of the input of the S-box). Therefore, in total, we get more than 20 active bytes at the input of layer 3 and output of layer 1. At the output of layer 1, the number of active bytes is usually a lower bound for the number of active S-boxes (due to the inverse mix column operation, which will rarely reduce the number of the active bytes, especially if the original differences are linear independent as assumed).

Now without loss of generality, assume that the number of active bytes at the output of layer two is larger or equal to the number of active bytes in the input, i.e., there are at least 10 active bytes in the input of the third layer, and at least 10 active S-boxes in the third layer. We can then safely assume that with a very high probability a majority of the S-boxes at the fourth layer are active, and therefore, that almost all the 128 $rk[\cdot]$ bytes generated after that layer are active.

We therefore conclude that except for a negligible probability, the number of active bytes in $rk[\cdot]$ is much higher than $1 + 20 + 32 + 128 = 181$. When counting the active bytes generated between the second and third layer, and between the third and the fourth, which are not directly an input to the AES layers, the total number of active bytes in $rk[\cdot]$ is expected to be way over 200.

We therefore conclude that either the attacker make trial hashing of a huge number of messages in order to get one with fewer than 200 active bytes — and this process will be very time consuming, or that the 200 or more active bytes in $rk[\cdot]$ will affect the computation of the main function with 200 active 'hits', each of them may be canceled at random with probability 2^{-8} on average, i.e., the probability of any characteristic may not be over 2^{-1600}. As the total size of the input to the compression function is 512+256+256+64=1088 bits, we expect that

there is not even a single right pair for almost all the characteristics, and a very small number of right pairs for the rest. Under these circumstances, no differential attack may be performed.

6.1.3 Resistance to (Second) Preimage Attacks on the Compression Function The use of Davies-Meyer makes the inversion of the compression function impossible without weaknesses in the block cipher. As analyzed before, the block cipher is secure, and thus no such attacks are feasible.

In some instances, collision attacks can be transformed for a second preimage attacks for a set of "weak messages", i.e., messages which satisfy some collision producing differential. As we showed before, there are none of these, and thus there are no such classes of weak message, i.e., the second preimage resistance is optimal.

6.1.4 Security in the Presence of Multiple Salts Our previous analysis assumed that the adversary has control over the salt, but has to choose the same salt for both message blocks. Repeating the previous analysis when the adversary has more control (additional 256-bit freedom in SHAvite-3_{256} and 512 more bits of freedom in SHAvite-3_{512}), reveals that both hash functions are still secure, as difference in the salt may cancel message differences when $rk[\cdot]$ and the salt have an active column in the same position. On the other hand, when $rk[\cdot]$ has no active column, the salt difference leads to an active column.

In other words, if the adversary uses two different salts, then the evolution of differences in the message expansion is faster. Bytes that earlier were not active, get activated by the salt difference (bytes can also become inactive with some probability/lose of freedom).

Thus, even in the presence of multiple salts, the security properties of collision resistance and (second) preimage resistance, are preserved.

6.2 The Security of HAIFA

The HAIFA mode of iteration offers security against many attacks. As noted earlier, HAIFA maintains the security of the compression function. The standard security features which are preserved, as well as the more advanced properties, make SHAvite-3 a secure candidate.

The above claim might seem contradictory to the results obtained in [2], which claim that HAIFA does not preserve various properties of the compression function. The proof of [2] is based on constructing a special compression function which posses undesired and unrealistic properties, and using these properties to attack the hash function (despite the security of the compression function). The compression functions we use are strong, and do not posses the weaknesses used in the special construction (or any weakness for that matter). Moreover, recent results obtained in [17], show that if the compression function is secure (i.e., is a fixed input length random oracle), then there are no shortcut second preimage attacks on the hash function. Thus, we conclude that SHAvite-3 is a secure hash function.

Both HAIFA and SHAvite-3 ensure that there are no related-salts issues. The best way to find a pair of messages and salts $(M, salt)$ and $(M', salt')$ such that SHAvite-$3_{salt}(M) =$ SHAvite-$3_{salt'}(M')$ (or that satisfy any other relation) is best achieved by generic attacks, e.g., the birthday attack. Moreover, given a digest y, the best approach to find a pair of message and salt $(M, salt)$ such that SHAvite-$3_{salt}(M) = y$ requires the use of generic attacks (i.e., exhaustive search).

Another security property that HAIFA hash functions (and thus SHAvite-3) posses is the lack of extension attacks. While for many iterated constructions $h(x||z)$ can be derived from

$h(x)$ and z, without even knowing x, in HAIFA this is impossible. The reason for that is the way the last block (or the last two blocks, in case an additional padding block is added) is treated. In the last block, the compression function is called with the number of bits that were processed so far. If this value is not a multiple of a block, then the resulting chaining value is not equal to the chaining value that is needed in case the message is extended. If the message is a multiple of a block, then an additional block is processed (with the parameter $\#bits$ set to 0). Thus, the chaining value required for the extended message remains unknown to the adversary.

As noted earlier, HAIFA maintains the collision resistance of the compression function. The underlying compression functions of SHAvite-3 are strong under the assumption that the block ciphers used are secure (which is the case), and thus SHAvite-3 offers a secure pseudorandom oracle and pseudorandom function behavior (up to the birthday bound, or more precisely, up to $\min\{2^m, 2^{128}\}$ for digests of length $m \leq 256$ bits, or up to 2^{256} for digests of length $257 \leq m \leq 512$).

6.3 Security of the Constructions Using SHAvite-3

As SHAvite-3 is a secure hash function, each construction using it in a "sane" manner is expected to be secure. This is also true for the signatures schemes and HMAC. Moreover, SHAvite-3 offers an inherent secure mode for randomized hashing through the use of salts.

6.3.1 Security of Signature Schemes As SHAvite-3 is a collision resistant and second preimage resistance hash function, it can be used in secure signature schemes. SHAvite-3 can replace SHA-1, any of the SHA-2 family, or any other used hash function (which provides digests of length up to 512 bits). Explicitly, SHAvite-3 can be used in any of these constructions with the fixed salt.

In applications where the salt can be communicated as well, e.g., the Digital Signature Standard (DSS) [50], one could use the randomness as the salt as well (or part of the salt). As the use of salts increases the security of the hash function (just like in randomized hashing [29]), we suggest new signature schemes to allow for a mechanism to communicate the salt.

6.3.2 Support for Randomized Hashing in SHAvite-3 The main purpose of randomized hashing is to reduce the level of requirements from the compression function in order to achieve more secure hash function [29]. The randomized hashing is especially useful for digital signatures, where the additional random inputs allow for a weaker compression function to be used. This even motivated NIST to put forward a special publication on the matter [53].

In [29] two constructions offering better security for hash functions are presented (and proved to be secure). While these proofs show that randomized hashing increases the security of unsalted hash functions (i.e., SHAvite-3 with fixed salts), we believe that better security can be achieved by just using the random value as the salt in a simple SHAvite-3 call.

6.3.3 Security of HMAC-SHAvite-3 HMAC's security is based on the pseudorandomness of the underlying compression function [5]. We recall that HAIFA preserves the pseudorandomness of the compression function and that C_{256} and C_{512} are secure when keyed by a random salt. Hence, we conclude that SHAvite-3 offers a secure base for HMAC.

In Section 7 we present a message authentication code which offers the same security as HMAC based on SHAvite-3, while offering better performance. Thus, we suggest that users of SHAvite-3 would use the more efficient construction.

7 HAIFA-MAC and SHAvite-3-MAC

According to the security analysis, HAIFA hash functions are protected against extension attacks and offer PRF preservation of the compression functions. Thus, unlike unsalted constructions, if the salt of the compression function is treated in a strong manner[2] than it is possible to define a secure HAIFA-MAC. HAIFA-MAC using the compression function $C(\cdot)$ and the key k is defined as:

$$\text{HAIFA-MAC}_k(M) = \text{HAIFA}_k^C(M).$$

Given the PRF preservation of HAIFA, then the above construction is a secure MAC if the compression function $C(\cdot)$ is a PRF. Thus, it is possible to replace more complex hash-based message authentication codes with a simple instance of HAIFA.

Moreover, the different IV_m for different digest sizes (and different tag sizes), as well as the encoding of the digest (tag) size in the padding of the message, ensure that even under the same key, the same message have completely different and uncorrelated digests (tags) of different lengths.

In order to offer true security, the compression function $C(\cdot)$ has to be indeed a pseudorandom function (and preferably related-key pseudorandom function). As SHAvite-3 has a secure compression function, we define

$$\text{SHAvite-3-MAC}_k(M) = \text{SHAvite-3}_k(M).$$

We note that for tags of up to 256 bits, one should use SHAvite-3_{256} as the hash function, while for tags of longer tags (up to 512 bits), one should use SHAvite-3_{512}.

The efficiency of SHAvite-3-MAC is better than of HMAC-SHAvite-3. Consider a message M, and a tag of size up to 256 bits. Computing SHAvite-3-MAC$_k(M)$ takes $\lceil |M| + 81/512 \rceil$ compression function calls and requires one initialization (including one initialization of the key). When computing HMAC-SHAvite-3 of the same message, the compression function is called $\lceil |M| + 81/512 \rceil + 1$ times, and there are two initializations. Even if the hash function in use does not add the tag size to the last block, the number of calls to the compression function in HMAC is $\lceil |M| + 65/512 \rceil + 1$, which in most cases is still one more call to the compression function.

The performance advantage may seem small (one compression function call), but for short messages (up to 53 bytes), it is 50% gain (also in the number of initializations), and for messages of 1500 bytes (a very common message size) the gain is 4% in the number of compression function calls.

8 Performance

SHAvite-3 is well suited for various platforms and machines, just like the AES. The byte-oriented structure and the AES building blocks, make SHAvite-3 "native" on 8-bit machines, 32-bit machines, 64-bit machines, and actually any machine that already supplies or uses AES.

The running times of our current (slightly optimized) ANSI-C code is 35.3 cycles per byte for 224-/256-bit digests, and 58.4 cycles per byte for 384-/512-bit digests on 32-bit Intel machines. On a 64-bit machine, the corresponding running times are 26.7 and 38.2 cycles per

[2] HAIFA does not define exactly in which manner the salt has to be mixed. However, our intentions are that the salt is mixed appropriately, offering a true effect of the salt on the compression function. For example, some weak possibilities are identified and discussed in [4].

byte, respectively. The code uses a relatively simple optimization techniques for AES, but does not use any special assembly or extended instruction sets, and thus well-optimized SHAvite-3 implementations are expected to be much faster.

We note that these numbers are based on a general 32-bit/64-bit machines architecture. Once the AES instruction set will be added to the Intel CPUs (expected in the second quarter of 2009), these speeds will improve significantly, as instead of performing an AES round in 21–29 cycles (the best known speed at the moment on common CPUs), the speed of an AES round would be reduced to roughly 6 cycles. Of course, not all of the speed up can be "exploited", but it is reasonable to assume that SHAvite-3 would enjoy at least 60% speed increase, and arguably even more.

8.1 Software Implementation Ideas

8.1.1 8-Bit Machines Just like AES, SHAvite-3 is highly suitable for 8-bit machines, and by using a table lookup for the S-box, a straightforward implementation of SHAvite-3 is possible. For the implementation of the MixColumns operation (the only non-byte operation) one can use the same suggestion as in [20]:

> "The only field multiplication used in this algorithm is multiplication with the element 02, denoted by 'xtime'.
>
> $t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; \quad /* \, a \text{ is a column} \, */$
> $u = a[0];$
> $v = a[0] \oplus a[1]; \quad v = xtime(v); \quad a[0] = a[0] \oplus v \oplus t;$
> $v = a[1] \oplus a[2]; \quad v = xtime(v); \quad a[1] = a[1] \oplus v \oplus t;$
> $v = a[2] \oplus a[3]; \quad v = xtime(v); \quad a[2] = a[2] \oplus v \oplus t;$
> $v = a[3] \oplus u; \quad v = xtime(v); \quad a[3] = a[3] \oplus v \oplus t;$"

SHAvite-3_{256} deploys 52 AES rounds, as well as about 192 32-bit XOR operations (which may be implemented by $192 \cdot 4 = 768$ 8-bit XORs). In [47] an AES-128 implementation which takes 3766 cycles per block is reported on an AVR processor. This speed is about 377 cycles per full AES round. Hence, the running time of SHAvite-3_{256} on AVR is expected to be about 20370 cycles per each invocation of the compression function, or about 318 cycles per byte. SHAvite-3_{512} uses 168 rounds of AES and 528 32-bit XOR operations, and thus the expected running time of C_{512} for each invocation is about 65450 cycles per byte, or a speed of 511 cycles per byte.

In [49] the 8-bit AVR core was extended with about 1100 gates, to reach speeds of 1300 cycles per 10-round AES encryption/decryption. This extension allows for implementing one round of AES in about 130 cycles. Thus, with this small extension, the speed of SHAvite-3 can be greatly improved to about 118 cycles per byte for digests of up to 256 bits, and 187 cycles per byte for digests of 257 to 512 bits.

8.1.2 32-Bit Machines For 32-bit machines, one can join together the accesses for the S-boxes along with the MixColumns operation, exploiting the linearity of the MixColumns operation. This approach requires the use of four tables, each containing 256 elements of 32 bits each (note that as we use only the full round, there is no need for the fifth table usually required for encryption in this approach). We note that we can also embed the salt into the tables (by having several instances of the tables), which would save the XOR of the salt in exchange for an additional memory.

One can use some of the suggestions in [3, 9] to speed up assembly implementations of AES (and thus of SHAvite-3), using CPU-specific instructions. The exact performance is hard to predict, but we expect that better coding practices, and the use of assembly would improve our current speed of 35.3 cycles per byte on a 32-bit machine (AMD Sempron(tm) Processor 3200+, 1800 MHz, 128 KB cache, 1 GB RAM, running in a full 32-bit mode).

On a different 32-bit machine, Intel Pentium4 f12, Bernstein and Schwabe report a speed of 14.13 cycles per byte for 10-round AES (in counter mode) [9]. While encryption in counter mode can be easily parallelized, it seems that it is a valid assumption that the speed of 10-round AES implementation of ECB on this machine can reach speeds of less than 18 cycles per byte (or $18 \cdot 16 = 288$ cycles in total) which are about 29 cycles per round. SHAvite-3 with digests of up to 256 bits uses 52 AES rounds, as well as about 192 32-bit XOR operations. Hence, we estimate a fully optimized code for this particular machine to achieve speeds of about 1700 cycles in total, or slightly less than 26.6 cycles per byte. SHAvite-3_{512} has a running time of 55.0 cycles per byte on the same Sempron machine, where it seems that a more optimized code may achieve speeds of about 5420 cycles per block which are 42.3 cycles per byte.

For comparison, on the same machine, which we obtained 35.3 cycles per byte for (not-well-optimized) SHAvite-3_{256}, the fastest SHA-1 implementation has a running time of 9.8 cycles per byte, SHA-256 had a running time of 28.8 cycles per byte, and SHA-512 had a running time of 77.8 cycles per byte. All measurements were done using the NESSIE test suite [44].

8.1.3 64-Bit Machines There are several approaches to implement AES on 64-bit machines. The first one follows the previously mentioned improvements and optimizations, while taking into consideration the larger number of registers and commands available on newer machines. A different approach is the use of of bit-sliced implementation proposed in [11], which were applied in [40, 41] to implement AES efficiently. Even though these implementations claim record speeds of less than 10 cycles per byte for 10-round AES, they are unsuitable for SHAvite-3, as bit-sliced approach is well suited for independent executions (which is not the case for SHAvite-3).

On the other hand, it seems that the speed of AES on 64-bit machine can reach 10.5 cycles per byte in counter mode, even without bit-slicing [9]. Thus, we assume that a fully optimized AES implementation can reach the speed of 13 cycles per byte, or about 21 cycles per round, on 64-bit machines. Thus, it is estimated that C_{256} would require about 1200–1300 cycles for each call, i.e., a speed of 18.6–20.3 cycles per byte should be reachable in a fully optimized code. Similar analysis for SHAvite-3_{512} reveals prospective speed of about 28.4–31.8 cycles per byte in an optimized implementation.

At the moment, our C-implementation of SHAvite-3_{256} has a running time of 26.7 cycles per byte on AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ (2200 MHz, 512 KB cache, 1 GB RAM). The code of SHAvite-3_{512} has a running time of 38.2 cycles per byte. For comparison, on this machine, SHA-1 takes 9.5 cycles per byte, SHA-256 takes 25.3 cycles per byte, and SHA-512 takes 16.9 cycles per byte.

8.2 Future Platforms

It is evident that adding the set of AES commands to Intel CPUs is expected to speed up AES implementations as well as SHAvite-3's implementations. The expected latency of this command is 6, i.e., it would take 6 cycles to perform one AES round [32]. Also, the platform is expected to allow multiple calls for the command (i.e., it is possible to compute two independent AES rounds in parallel within 7 cycles).

Hash Function	32-Bit Machine	64-Bit Machine
MD5	7.4	8.8
SHA-1	9.8	9.5
SHA-256	28.8	25.3
SHA-512	77.8	16.9
SHAvite-3_{256} (measured)	35.3	26.7
SHAvite-3_{256} (conjectured)	26.6	18.6
SHAvite-3_{256} (with AES inst.)	< 8	
SHAvite-3_{512} (measured)	55.0	38.2
SHAvite-3_{512} (conjectured)	35.3	28.4
SHAvite-3_{512} (with AES inst.)	< 12	

Table 4. Speed Comparison of Hash Functions (in cycles/byte)

With such a command, and sufficient number of registers, we expect that the speed of SHAvite-3_{256} could be improved to a total of about 500 cycles per each invocation of C_{256} without applying AES rounds in parallel, and much faster when independent AES rounds may be performed in parallel. This would lead to a running time of less than 8 cycles per byte on such CPUs.

For SHAvite-3_{512}, where we expect even better use of the command, non-interleaved code (with 168 AES rounds and 528 32-bit XORs) is expected to have a running time of 1540 cycles per invocation, or 12 cycles per byte. However, this figure is an overestimation, as the calls for the AES round instructions themselves can be interleaved.

8.3 Hardware Implementations of SHAvite-3

AES is well suited for hardware platforms such as Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Arrays (FPGAs). SHAvite-3 implementations are expected to be very similar to an AES implementation, up to the larger internal state, and the XORs used in the Feistel construction. Thus, we first summarize some performance results on AES, and then predict the expected hardware efficiency.

AES was implemented in many ways and manners, trying to optimize for various goals. We shall be interested in two main optimization goals: size (gates or slices) and speed. Hence, we are interested in each of the two technologies in order to estimate the size and speed of SHAvite-3 implementations in them.

8.3.1 Implementations of AES in Hardware The smallest AES implementation in ASIC is reported in [27]. The suggested implementation uses about 3400 gates, and has a throughput of 9.9 Mbps in a 80 MHz maximal frequency (the implementation used a 0.35μ technology). Of the 3400 gates, about 60% (about 2040) are reported to store 256 bits of the internal state (a rate of about 8 gates per memory bit).

While there are only a few papers on fast ASIC implementations of AES, they try to achieve high throughput by using many pipelined application of AES. This, of course, can only work if the cipher is used in ECB or CTR modes of operation. With one pipeline per round (i.e., 10 encryptions in parallel), the results of [31] are a throughput of about 44 Gbps using slightly less than 250,000 gates. In this implementation, the round function is implemented 10 times, and the subkeys are fixed. Hence, there are about $128 \cdot 11$ memory bits containing subkeys

and additional $128 \cdot 10$ memory bits to store intermediate encryption values. The total of 2688 memory bits take about 21,500 gates, giving an estimate of 228500 gates for fully implemented 10-round AES and the combining logic around it. Hence, we assume that the cost of a (very fast) AES implementation in hardware is about 22,850 gates per round running on a 0.18μ technology at about 340 MHz.

For FPGA implementations, we consider again the two optimization targets. In [28], an implementation of AES on a Spartan-II FPGA is reported to take a total of 264 slices (both for the data and the memory), of which 124 slices compose the actual encryption process, and the remaining 140 slices contain the memory. The throughput of this implementation is 2.2 Mbps in a 67 MHz clock frequency.

The fastest FPGA implementation we could locate was the one of [59]. The implementation reaches speeds of 23.57 Gbps (in ECB/CTR mode) with 16398 slices running at speed of 184.16 MHz. The latency is 162.9 nano-second, i.e., it takes 30 cycles to encrypt a 16-byte block.

8.3.2 Implementing SHAvite-3_{256} in Hardware When implementing SHAvite-3_{256} in hardware using an AES implementation we need to consider several factors:

- The memory consumption of C_{256} is larger. The implementation stores 512-bit message register (containing 16 words of the expanded message), 256-bit salt, 64-bit counter, 256-bit input chaining value, and 256-bit intermediate compression value.
- There is a need for three consecutive AES rounds in $F^3(\cdot)$.
- The message expansion can be computed four words at a time.

Hence, an area efficient approach would implement the AES round once, and use it repeatedly for each application. Due to the way SHAvite-3 works, the implementation based on [27] would need another 128-bit of internal state to store intermediate results of the AES round. Hence, we estimate that the implementation would require about 8832 gates for storing all the internal state bits (assuming each bit requires about 6 gates), and another 1360 gates for the AES core,[3] along with about 100 more gates for the XORs and control overhead. This results in a full implementation of SHAvite-3_{256} in about 10300 gates. We note that some of these gates are part of memory that may be stored outside the core of the hash function (e.g., the 64-bit counter can be stored in a different area, which probably would cost less gates in the core of the compression function).

The speed of this implementation is about 100 cycles for an AES round (at 80 MHz), which implies a speed of about 5200 cycles for an invocation of C_{256}, or a throughput of about 7.6 Mbps.

For the fast ASIC implementation we consider a similar methodology, but using the different implementation figures of [31]. First of all, we note that SHAvite-3_{256} can be implemented using only two AES-round cores (rather than 10), as in any case there are at most two AES rounds occurring at the same time (one in $F^3(\cdot)$ and one in the message expansion). Thus, besides storing the same amount of bits as in the small area size implementation (which takes 8832 gates), we need the two AES round implementations (each takes 22,850 gates), and an overhead of about 400 gates for the XOR and additional control area (the control in high speed environments is usually larger). The total gate count is therefore expected to be about 55,000 gates. The speed of an AES round is one cycle in this implementation. The longest datapath

[3] We note that this core also contains the key schedule circuit, which can be omitted for SHAvite-3. However, its size is relatively small, which can be approximated as zero when estimating the total circuit size.

of C_{256} is of 36 rounds of AES, and thus, it seems that one invocation of C_{256} takes about 36 cycles. Hence, using 55,000 gates, the expected throughput of this implementation is 604.4 Mbps.

We can apply exactly the same analysis to FPGA implementations. The smallest FPGA implementation of AES uses 124 slices for the AES round, and 70 slices to store 256 bits of internal state. Hence, we estimate that 385 slices would be sufficient to store all the data for C_{256}, and thus, along with the 124 slices of the AES round, we expect a total of about 510 slices for a full implementation of SHAvite-3_{256}. The speed of the FPGA implementation is about 3900 cycles per each 10-round AES call, or about 390 cycles for one round. Hence, the speed for one call to C_{256} is expected to be about 20,300 cycles, or a throughput of 1.7 Mbps.

When analyzing the above mentioned fast FPGA implementation, we can see that the are about 30 instances of AES running in parallel, which takes quite a lot of memory. We shall assume that of the 16398 slices, 16000 are used for the logic and only 400 are used as memory (this is an overestimation of the logic). Hence, one AES round can be implemented using 1600 slices in such a way that it takes three cycles to compute. Using the same ideas as for the fast ASIC implementation, we expect 3200 slices for the two rounds of AES and about 385 more slices for the memory. We conclude that this implementation is expected to use about 3585 slices, and takes 108 cycles for each compression function call (i.e., a throughput of 872.3 Mbps).

8.3.3 Implementing SHAvite-3_{512} in Hardware Applying the same methodology as for SHAvite-3_{256} to reduced size ASIC implementation, we obtain the following estimations: The implementation needs to store 2816 bits (128 bits for the AES state, 1024 bits for the message block, two 512-bit registers for the chaining value (before and after E^{512}) and 128 bits for the counter). Hence, the implementation is expected to use about 18500 gates, and achieve a speed of about 4.7 Mbps.

When implementing C_{512} targeting a fast implementation in ASIC using the methodology described in [31], four AES round cores need to be used. This increases the circuit size to about 100,500 gates. As the longest datapath has 48 rounds of AES, we expect a throughput of about 907.7 Mbps.

For a small area FPGA implementation, the expected size is 895 slices (the difference is due to the additional internal memory), and the expected throughput of about 1.0 Mbps. The fast FPGA implementation is expected to use 7170 slices and achieve speeds of 168 cycles for a compression function call (which means a throughput of 1121.5 Mbps).

9 Summary

In this document we have presented SHAvite-3, a new efficient and secure hash function. We devised SHAvite-3 with large security margins in order to ensure security for years to come. At the same time, we have also considered efficiency of both software and hardware implementations.

We would like to thank the following people: Charles Bouillaguet, Yaniv Carmeli, Rafi Chen, Pierre-Alain Fouque, Nicolas Gama, Edmond Halley, Sebastiaan Indesteege, Nathan Keller, Gaëtan Leurent, Osnat Ordan, Adi Shamir, and Frederik Vercauteren. Our discussions with them, as well as their good advice and heritage, made SHAvite-3 a better hash function.

References

1. Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, Sébastien Zimmer, *Second preimage attacks on dithered hash functions*, Advances in Cryptology, proceedings of EUROCRYPT 2008, Lecture Notes in Computer Science 4965, pp. 270–288, Springer-Verlag, 2008.
2. Elena Andreeva, Gregory Neven, Bart Preneel, Thomas Shrimpton, *Seven-Properties-Preserving Iterated Hashing: ROX*, Advances in Cryptology, proceedings of ASIACRYPT 2007, Lecture Notes in Computer Science 4833, pp. 130–146, Springer-Verlag, 2007.
3. Kazumaro Aoki, Helger Lipmaa, *Fast Implementations of AES candidates*, proceedings of the third AES conference, pp. 106–120, New York, 2000.
4. Jean-Philippe Aumasson, Raphael C.-W. Phan, *How (Not) to Efficiently Dither Blockcipher-Based Hash Functions?*, proceedings of AFRICACRYPT 2008, Lecture Notes in Computer Science 5023, pp. 308–324, Springer-Verlag, 2008.
5. Mihir Bellare, *New Proofs for NMAC and HMAC: Security Without Collision-Resistance*, Advances in Cryptology, proceedings of CRYPTO 2006, Lecture Notes in Computer Science 4117, pp. 602–619, Springer-Verlag, 2006.
6. Mihir Bellare, Ran Canetti, Hugo Krawczyk, *Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security*, proceedings of 37th Annual Symposium on Foundations of Computer Science (FOCS '96), pp. 514–523, IEEE Computer Society, 1996.
7. Mihir Bellare, Thomas Ristenpart, *Multi-Property-Preserving Hash Domain Extension: The EMD Transform*, Advances in Cryptology, proceedings of ASIACRYPT 2006, Lecture Notes in Computer Science 4284, pp. 299–314, Springer-Verlag, 2006.
8. Mihir Bellare, Philip Rogaway, *Collision-resistant hashing: Towards making UOWHFs practical*, Advances in Cryptology, proceedings of CRYPTO 1997, Lecture Notes in Computer Science 1294, pp. 470–484, Springer-Verlag, 1997.
9. Daniel J. Bernstein, Peter Schwabe, *New AES software speed records*, IACR ePrint report 2008/381.
10. , Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, *On the Indifferentiability of the Sponge Construction*, Advances in Cryptology, proceedings of EUROCRYPT 2008, Lecture Notes in Computer Science 4965, pp. 181–197, Springer-Verlag, 2008.
11. Eli Biham, *A Fast New DES Implementation in Software*, proceedings of Fast Software Encryption 1997, Lecture Notes in Computer Science 1267,pp. 260–272, Springer-Verlag, 1997.
12. Eli Biham, Rafi Chen, *Near-Collisions of SHA-0*, Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152, pp. 290–305, Springer-Verlag, 2004.
13. Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, William Jalby, *Collisions of SHA-0 and Reduced SHA-1*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3621, pp. 36–57, Springer-Verlag, 2005.
14. Eli Biham, Orr Dunkelman, *A Framework for Iterative Hash Functions — HAIFA*, NIST 2nd hash function workshop, Santa Barbara, August 2006.
15. Eli Biham, Orr Dunkelman *A Framework for Iterative Hash Functions - HAIFA*, IACR ePrint report 2007/278.
16. Eli Biham, Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.
17. Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Sébastien Zimmer, *(Revisiting the) Second Preimage Resistance of Some Iterated Hash Functions*, preprint, 2008.
18. Florent Chabaud, Antoine Joux, *Differential Collisions in SHA-0*, Advances in Cryptology, proceedings of CRYPTO 1998, Lecture Notes in Computer Science 1462, pp. 56–71, Springer-Verlag, 1998.
19. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, Prashant Puniya, *Merkle-Damgård Revisited: How to Construct a Hash Function*, Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 430–448, Springer-Verlag, 2005.
20. Joan Daemen, Vincent Rijmen *The design of Rijndael: AES — the Advanced Encryption Standard*, Springer-Verlag, 2002.

21. Joan Daemen, Gilles Van Assche, *Producing Collisions for Panama, Instantaneously*, proceedings of Fast Software Encryption 2007, Lecture Notes in Computer Science 4593, pp. 1–18, Springer-Verlag, 2007.
22. Ivan Damgård, *A Design Principle for Hash Functions*, Advances in Cryptology, proceedings of CRYPTO 1989, Lecture Notes in Computer Science 435, pp. 416–427, Springer-Verlag, 1990.
23. Christophe De Cannière, Christian Rechberger, *Finding SHA-1 Characteristics: General Results and Applications*, Advances in Cryptology, proceedings of ASIACRYPT 2006, Lecture Notes in Computer Science 4284, pp. 1–20, Springer-Verlag, 2006.
24. Christophe De Cannière, Christian Rechberger, *Preimages for Reduced SHA-0 and SHA-1*, Advances in Cryptology, proceedings of CRYPTO 2008, Lecture Notes in Computer Science 5157, pp. 179–202, Springer-Verlag, 2008.
25. Richard D. Dean, *Formal Aspects of Mobile Code Security*, Ph.D. dissertation, Princeton University, 1999.
26. ECRYPT, *State of the Art in Hardware Architectures*, report D.VAM.2, September 2005, available online at *http://www.ecrypt.eu.org/documents.html*.
27. Martin Feldhofer, Johannes Wolfkerstorfer, Vincent Rijmen *AES implementation on a grain of sand*, IEE Proceedings of Information Security, Vol. 152, No. 1, pp. 13–20, IEE, 2005.
28. Tim Good, Mohammed Benaissa, *AES on FPGA from the Fastest to the Smallest*, proceedings of Cryptographic Hardware and Embedded Systems — CHES 2005, Lecture Notes in Computer Science 3659, pp. 427–440, Springer-Verlag, 2005.
29. Shai Halevei, Hugo Krawczyk, *Strengthening Digital Signatures via Randomized Hashing*, Advances in Cryptology, proceedings of CRYPTO 2006, Lecture Notes in Computer Science 4117, pp. 41–59, Springer-Verlag, 2006.
30. Jonathan J. Hoch, Adi Shamir, *Breaking the ICE — Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions*, proceedings of Fast Software Encryption 2006, Lecture Notes in Computer Science 4047, pp. 199–214, Springer-Verlag, 2006.
31. Alireza Hodjat, Ingrid Verbauwhede, *Minimum Area Cost for a 30 to 70 Gbits/s AES Processor*, proceedings of IEEE computer Society Annual Symposium on VLSI, pp. 83–88, IEEE, 2004.
32. Intel, *Advanced Encryption Standard (AES) Instructions Set*, white paper, July 2008. Available online at *http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf*.
33. Antoine Joux, *Multicollisions in Iterated Hash Functions*, Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152, pp. 306–316, Springer-Verlag, 2004.
34. Antoine Joux, Thomas Peyrin, *Hash Functions and the (Amplified) Boomerang Attack*, Advances in Cryptology, proceedings of CRYPTO 2007, Lecture Notes in Computer Science 4622, pp. 244–263, Springer-Verlag, 2007.
35. Liam Keliher, Jiayuan Sui, *Exact Maximum Expected Differential and Linear Probability for 2-Round Advanced Encryption Standard (AES)*, IACR ePrint report 2005/321.
36. John Kelsey, Tadayoshi Kohno, *Herding Hash Functions and the Nostradamus Attack*, Advances in Cryptology, proceedings of EUROCRYPT 2006, Lecture Notes in Computer Science 4004, pp. 183–200, Springer-Verlag, 2006.
37. John Kelsey, Bruce Schneier, *Second Preimages on n-Bit Hash Functions for Much Less than* 2^n, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 474–490, Springer-Verlag, 2005.
38. Jongsung Kim, Seokhie Hong, Jaechul Sung, Changhoon Lee, Sangjin Lee, *Impossible Differential Cryptanalysis for Block Cipher Structures*, proceedings of INDOCRYPT 2003, Lecture Notes in Computer Science 2904, pp. 82–96, Springer-Verlag, 2003.
39. Stefan Lucks, *A Failure-Friendly Design Principle for Hash Functions*, Advances in Cryptology, proceedings of ASIACRYPT 2005, Lecture Notes in Computer Science 3788, pp. 474–494, Springer-Verlag, 2005.
40. Mitsuru Matsui, *How Far Can We Go on the x64 Processors?*, proceedings of Fast Software Encryption 2006, Lecture Notes in Computer Science 4047, pp. 341–358, Springer-Verlag, 2006.
41. Mitsuru Matsui, Junko Nakajima, *On the Power of Bitslice Implementation on Intel Core2 Processor*, proceedings of Cryptographic Hardware and Embedded Systems — CHES 2007, Lecture Notes in Computer Science 4727, pp. 121–134, Springer-Verlag, 2007.

42. Ralph C. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research press, 1982.
43. Ralph C. Merkle, *One Way Hash Functions and DES*, Advances in Cryptology, proceedings of CRYPTO 1989, Lecture Notes in Computer Science 435, pp. 428–446, Springer-Verlag, 1990.
44. NESSIE, *The NESSIE Test Suite*, version 3.1.1, 2002.
45. Luke O'Connor, *On the Distribution of Characteristics in Bijective Mappings*, Advances in Cryptology, proceedings of EUROCRYPT'93, Lecture Notes in Computer Science 765, pp. 360–370, Springer-Verlag, 1994.
46. Thomas Peyrin, *Cryptanalysis of Grindahl*, Advances in Cryptology, proceedings of ASIACRYPT 2007, Lecture Notes in Computer Science 4833, pp. 551–567, Springer-Verlag, 2007.
47. Sören Rinne, Thomas Eisenbarth, Christof Paar, *Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers*, 3rd International Symposium on Industrial Embedded Systems — SIES 2008, pp. 58–66, 2008, available online at *http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/lw_speed2007.pdf*.
48. Phillip Rogaway, Thomas Shrimpton, *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance*, proceedings of Fast Software Encryption 2004, Lecture Notes in Computer Science 3017, pp. 371–388, Springer-Verlag, 2004.
49. Stefan Tillich, Christoph Herbst, *Boosting AES Performance on a Tiny Processor Core*, proceedings of CT-RSA 2008, Lecture Notes in Computer Science, 4964, pp. 170–186, Springer-Verlag, 2008.
50. US National Institute of Standards and Technology, *Digital Signature Standard*, Federal Information Processing Standards Publications No. 186-2, 2000.
51. US National Institute of Standards and Technology, *Advanced Encryption Standard*, Federal Information Processing Standards Publications No. 197, 2001.
52. US National Institute of Standards and Technology, *Secure Hash Standard*, Federal Information Processing Standards Publications No. 180-2, 2002.
53. US National Institute of Standards and Technology, *Randomized Hashing for Digital Signatures*, Draft NIST Special Publication 800-106, 2008.
54. Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, Xiuyuan Yu, *Cryptanalysis of the Hash Functions MD4 and RIPEMD*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 1–18, Springer-Verlag, 2005.
55. Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Finding Collisions in the Full SHA-1*, Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 17–36, Springer-Verlag, 2005.
56. Xiaoyun Wang, Hongbo Yu, *How to Break MD5 and Other Hash Functions*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 19–35, Springer-Verlag, 2005.
57. Xiaoyun Wang, Hongbo Yu, Yiqun Lisa Yin, *Efficient Collision Search Attacks on SHA-0*, Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 1–16, Springer-Verlag, 2005.
58. Gideon Yuval, *How to Swindle Rabin*, Cryptologia, Vol. 3, pp. 187–190, 1979.
59. Joseph Zambreno, David Nguyen, Alok N. Choudhary, *Exploring Area/Delay Tradeoffs in an AES FPGA Implementation*, proceedings of Field Programmable Logic and Application (FPL) 2004, Lecture Notes in Computer Science 3203, pp. 575–585, Springer-Verlag, 2004.

A Test Vectors

A.1 Digests of 224-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-$3_0(M)$)
""	7D9F1D40 5B2663DA FE1C0C7B 1C6C19BC 94E9CBE0 BD9565DC B47FB00B$_x$
"A"	626F3FEA 5BEBF552 46826296 35C346ED DA6A20A7 877CFEEC A3BECCB0$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	105A1722 3F2DCD42 791F5113 18F90882 5F01248F C8A8876D 6DDED7C6$_x$
"AAA...AA" (1,000,000 times)	C4AC2A56 8B30C5D2 C67780BB 4C81B21A BBE4615E A10824F1 4E348B2A$_x$

For the salt value $salt = 11111111\ 11111111\ \ldots\ 11111111_x$:

Message (M)	Digest (SHAvite-$3_{11111111\ \ldots\ 11111111_x}(M)$)
""	D41EDFA2 B2E691C6 77142337 E20F5B3E 79F9A23A 4B844E92 BA7BEA2F$_x$
"A"	36DBC6D5 D9C79135 F8D53B8 81A18E36 9C4CCD4E 579F0D4D F7512F21$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	1E091305 6921EA55 33F180FC 9B09DA1E 539F45C7 FFE9D9BA 36D396BE$_x$
"AAA...AA" (1,000,000 times)	71B0521F 406F5AF7 C146194B FD04F65F AC191D69 BD90C35 21537BEB$_x$

A.2 Digests of 256-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-$3_0(M)$)
""	40A74666 D7F02BFD 75625297 327F7738 2CE204BE C7D64938 C2BCBB9F 00458FE6$_x$
"A"	DC508FBB 8040A4AC 5990C1A0 52A1F011 1B08BAEC AAB22D26 3AFEA41F 424446D4$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	E8B23C1F 733DF1D4 F49EEBEC 17F82556 CA7229D8 7CA2DFE4 77D28C7B D041876A$_x$
"AAA...AA" (1,000,000 times)	5F6B9516 5CBBC531 9A5C769F B1D77389 DAD41F91 FDC9FAA9 4B762F48 ABF8BE5F$_x$

For the salt value $salt = 22222222\ 22222222\ \ldots\ 22222222_x$:

Message (M)	Digest (SHAvite-$3_{22222222\ \ldots\ 22222222_x}(M)$)
""	5E3170B6 6138DCF1 585EFE6D 9C392719 7B9A2BBE 84A095AF
	F059FE6E A21D7220$_x$
"A"	1AD2ACFD B6A10DE9 FC419AB0 FFF00737 08523206 20EC043A
	A96917C9 FFA713BE$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	
	CB8BD4CF 569F4ED4 4606DA74 A18B5530 49966F7F 30F113DA
	A2E9D901 07C38CFE$_x$
"AAA...AA" (1,000,000 times)	
	AEBF5F52 E106B0FD 9DC9203D C82CE0E1 9A329FA9 24570E5B
	7E613D63 FED2943A$_x$

A.3 Digests of 384-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-$3_0(M)$)
""	9F3140AC 7AB5BD6E BC2D0D82 9C8F2129 DF19A358 55D25106
	A683DA0C 62578A77 7653EB0A 8D7D672A 19A4FDF6 49A3D9F1$_x$
"A"	29CA127F B43666EA FC7AA9DF 0D39C572 0087A448 228A206E
	0F673351 6E36986B DC31B388 69B1CD20 1D7F36E4 069D80A8$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	
	D9801602 8CF45738 E3693462 3C116A6C 4563630D 4409E219
	FCA250C5 FA1BCDF2 A5CAB273 72076F5E EDBF0DA5 6FBC251E$_x$
"AAA...AA" (1,000,000 times)	
	395D1054 4977600E 024FB44D 27BFBBEB 09543303 8A540089
	C7C4943D D4DDD9B3 4031E492 0810579F 8E4F2CFA 43E44135$_x$

For the salt value $salt = 33333333\ 33333333\ \ldots\ 33333333_x$:

Message (M)	Digest (SHAvite-$3_{33333333\ \ldots\ 33333333_x}(M)$)
" "	2980CDDC 8DDC9FF8 69B4FC1F DE6AD53B 3F217C54 A5D3B782
	DFB16B0A 9E80F1D5 9E896F89 E3FC3EE1 C9D7218D 59634006$_x$
"A"	45513793 DD182058 1B3D0D04 B245EC88 53551592 D5E1E9C5
	08B25934 40534150 39FCED7E 6B6D5897 2CFB5D71 1E932948$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	
	B1C3EC47 17551167 C2022012 40FD286F 58D614B1 611DDBCB
	C2B56CBE 7D10F428 2E4D0070 8F9F5011 FB018AA5 31D0D4A4$_x$
"AAA...AA" (1,000,000 times)	
	91EB16DC 0E9EA4DF 631A198F 5DAB30AF CBE8FCB8 4DDCBA72
	8F6C5948 54CDD2F8 E0A56D0A 6861F360 4B29FB5D 0359F2DA$_x$

A.4 Digests of 512-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-$3_0(M)$)
""	D7675CC2 C6D9004D B9C66894 872D3BE2 BF53F932 9DD77F09
	89825AE1 BD3CBC07 E78E0B64 6E453AED 8708846A F65DE1ED
	978CBAF4 2242F571 7E66DF67 556E1526$_x$
"A"	2C90009C 4AA6AC77 24D69F76 2E90A7C8 4EC25814 0CA3A2F4
	831FC93A 40C171A9 98A15906 96D9BAD4 CC587F57 6B377D39
	57204E33 549C1D4A FA2458D4 EDF4D027$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	
	D7BB7A7D B0FD516F 7082BB81 4FAC2376 733A462B 4800E553
	E1872241 A0B1CC20 61B66339 6646CEB2 C52A904B E2FD7AA8
	CDD0B368 A7864C3A 17E41A92 7CC699A9$_x$
"AAA...AA" (1,000,000 times)	
	B2825FC1 4180E600 11E75DE7 595A71D8 0D1C7555 B6B946BD
	88A359B4 895F81FE 5C4FD7E1 8A9D4ACA 1FC5AE65 31B536D6
	8042B004 C9ADB703 55051471 A1992061$_x$

For the salt value $salt = 44444444\ 44444444\ \ldots\ 44444444_x$:

Message (M)	Digest (SHAvite-$3_{44444444\ \ldots\ 44444444_x}(M)$)
""	C4701B86 C58DE5EC 46F724A4 9D2DFD4D C82F6B65 FDA8DEE6
	FEDEFE04 956D0EAB 28F50B41 63E1C165 658CFD79 9D41F9DA
	1DA8FE6B 7BA2166E 20824731 193795EE$_x$
"A"	88BD7C5B 36C55498 5E7412FF EDEF4C87 EFB2161B F6275B8C
	56EBFFBB 2BE18E52 508D051D 49B74F96 77F58639 1EDAAE91
	982C91F4 384CEC47 99C6686D 9F0E074B$_x$
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"	
	A4C599A3 AE358DA4 3C2744BC A665A29A 61289DB7 44731A48
	A78F54DC E1378CD3 741E06E3 7AC87DBE 9326537D 3808F9DC
	F9A1AC5C 3689B8CD E815B2FA CC04C47D$_x$
"AAA...AA" (1,000,000 times)	
	4650F205 3C132994 FA83994B 11F6A2E1 937E41A8 26818580
	E3EF471E 816F461C 495A43E3 72154215 FD52651D 3480C000
	40D382A3 F6E9C459 BBBAEF7C F8C406E8$_x$

B Examples of the Execution of SHAvite-3

B.1 Digests of 224 Bits

B.1.1 One-Block Message We outline here the internal state of the computation of computing the 224-bit digest of the message M = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

The first step is the padding of the message, yielding the array of 32-bit words:

$$msg[0,\ldots,15] = \begin{array}{l}\texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655}\\ \texttt{00805A59 00000000 00000000 00000000 00000000 00000000}\\ \texttt{00000000 00D00000 00000000 00E00000}_x\end{array}$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	00805A59	00000000
8	00000000	00000000	00000000	00000000	00000000	00D00000	00000000	00E00000
16	C6DD21A7	F285D8E5	E9B2B0C3	12574719	EC42280E	6DDCC7C0	791301B7	627BFE5A
24	8F214B6D	0EBFA4A3	1A7062D4	01189D39	EC42280E	F0C34542	791301B7	575DED49
32	361E64E5	8B96D952	BEEF5D8A	244923FC	67D4F15C	D3339A4A	5D5A224B	05AF0F06
40	5C12D127	53E586E8	1FDF6DD2	5D0A4C1E	BFA7AEE6	EF1C2890	24194DA9	E8FA43AF
48	408C70F1	A6013B01	95EAE148	B5A08507	695D9A07	C29136BA	90716778	06794F1D
56	F420F41F	55465B95	B9CDAE50	ACAC32C2	A2EC42AC	CA7D33C6	97616F33	99C0F057
64	8AF14337	31605432	0C2A111F	3F51C630	583DCE35	CEBB27A5	AF20A148	5E448128
72	3A9BD3BA	FA66FADD	E7892F78	9637E178	588AB871	2DF41CBE	01568E4B	C14A4826
80	D7C15C5D	9BBE622A	F3DA41CE	D016F5FE	82BDD957	C971073D	B4D14B69	27D14DF0
88	2622BF8C	E8A19DFF	22A24A98	C0B40B2E	B4D68C41	B3700686	7F2AA18F	2067EEC3
96	64B15ADB	E494C3A5	D3BDAF0D	B4A7AF25	66291AF2	1ACCA830	0076E44C	41F85702
104	3CEE17BC	E8D779B3	635A1D9A	FC5A1C92	5C01F5F2	D02A1B1C	8370BD1D	7C661B31
112	C0AEC423	402B4C58	38D3C6C2	DAD5C9D1	051868B4	E3333D16	3A750469	7FBCFA48
120	083EC1EA	4AC93494	37EC3857	4BF39D5A	3D8F255D	A6EE834D	984AE6E7	5621948E
128	6640476E	D861AABF	6EF2524C	BC958EBF	DD79C20B	8DC16F5A	86E08AD6	A2C53843
136	85FFAEB0	CC29BE42	95290014	CE0C33EA	F1A69B1F	33C78359	5646D50D	A7870F91

All values are given in hexadecimal.

The value of IV_{224} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	D617833B 68EA6C8F FF3DF700 E5B807EF	6FDB4E75 F966482E 3B40F9B2 755891B2
	Input to $F^3(\cdot)$:	6FDB4E75 F966482E 3B40F9B2 755891B2
	Output from $F^3(\cdot)$:	CB811D69 8BED4AC7 B9280AAF CCE48A71
1	6FDB4E75 F966482E 3B40F9B2 755891B2	1D969E52 E3072648 4615FDAF 295C8D9E
	Input to $F^3(\cdot)$:	1D969E52 E3072648 4615FDAF 295C8D9E
	Output from $F^3(\cdot)$:	6154F074 F261AEAD FEF98832 87F89D76
2	1D969E52 E3072648 4615FDAF 295C8D9E	0E8FBE01 0B07E683 C5B97180 F2A00CC4
	Input to $F^3(\cdot)$:	0E8FBE01 0B07E683 C5B97180 F2A00CC4
	Output from $F^3(\cdot)$:	1B93BE6F 7E7522AA A294B5CD 0B87FA87
3	0E8FBE01 0B07E683 C5B97180 F2A00CC4	0605203D 9D7204E2 E4814862 22DB7719
	Input to $F^3(\cdot)$:	0605203D 9D7204E2 E4814862 22DB7719
	Output from $F^3(\cdot)$:	ACAFE35C 0A36BFBD 5E0E9D35 031F08D9
4	0605203D 9D7204E2 E4814862 22DB7719	A2205D5D 0131593E 9BB7ECB5 F1BF041D
	Input to $F^3(\cdot)$:	A2205D5D 0131593E 9BB7ECB5 F1BF041D
	Output from $F^3(\cdot)$:	887A9D2C A6CE1182 A3BF5DCC 9AB52FB4
5	A2205D5D 0131593E 9BB7ECB5 F1BF041D	8E7FBD11 3BBC1560 473E15AE B86E58AD
	Input to $F^3(\cdot)$:	8E7FBD11 3BBC1560 473E15AE B86E58AD
	Output from $F^3(\cdot)$:	C45738BE 76828595 8F8B10E3 63D47EC8
6	8E7FBD11 3BBC1560 473E15AE B86E58AD	667765E3 77B3DCAB 143CFC56 926B7AD5
	Input to $F^3(\cdot)$:	667765E3 77B3DCAB 143CFC56 926B7AD5
	Output from $F^3(\cdot)$:	158C878A 1EE88B23 70D0FD15 ED1327EC
7	667765E3 77B3DCAB 143CFC56 926B7AD5	9BF33A9B 25549E43 37EEE8BB 557D7F41
	Input to $F^3(\cdot)$:	9BF33A9B 25549E43 37EEE8BB 557D7F41
	Output from $F^3(\cdot)$:	3F2D7621 90836A55 F1A3AD9C B57DCDEC
8	9BF33A9B 25549E43 37EEE8BB 557D7F41	595A13C2 E730B6FE E59F51CA 2716B739
	Input to $F^3(\cdot)$:	595A13C2 E730B6FE E59F51CA 2716B739
	Output from $F^3(\cdot)$:	CCF1EAE8 E3C98C58 3016B4B1 9F01F396
9	595A13C2 E730B6FE E59F51CA 2716B739	5702D073 C69D121B 07F85C0A CA7C8CD7
	Input to $F^3(\cdot)$:	5702D073 C69D121B 07F85C0A CA7C8CD7
	Output from $F^3(\cdot)$:	C7302D3E FE5A65DB EDA79E74 9A0D75DE
10	5702D073 C69D121B 07F85C0A CA7C8CD7	9E6A3EFC 196AD325 0838CFBE BD1BC2E7
	Input to $F^3(\cdot)$:	9E6A3EFC 196AD325 0838CFBE BD1BC2E7
	Output from $F^3(\cdot)$:	914F446A 915AB3D6 81DAFA19 373D83BA
11	9E6A3EFC 196AD325 0838CFBE BD1BC2E7	C64D9419 57C7A1CD 8622A613 FD410F6D
	Input to $F^3(\cdot)$:	C64D9419 57C7A1CD 8622A613 FD410F6D
	Output from $F^3(\cdot)$:	AEB05406 28A41C66 5EA6E1CA E5CF68A1
12	C64D9419 57C7A1CD 8622A613 FD410F6D	30DA6AFA 31CECF43 569E2E74 58D4AA46

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$\begin{aligned} h_0 = & \texttt{D617833B 68EA6C8F FF3DF700 E5B807EF 6FDB4E75 F966482E 3B40F9B2 755891B2} \\ & \oplus \\ & \texttt{C64D9419 57C7A1CD 8622A613 FD410F6D 30DA6AFA 31CECF43 569E2E74 58D4AA46} \\ & = \\ h_1 = & \texttt{105A1722 3F2DCD42 791F5113 18F90882 5F01248F C8A8876D 6DDED7C6 2D8C3BF4} \end{aligned}$$

This value is truncated to the digest:

$$105A1722\ 3F2DCD42\ 791F5113\ 18F90882\ 5F01248F\ C8A8876D\ 6DDED7C6_x.$$

B.1.2 Two-Block Message For a two block message, we picked the message

M = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq
rstuvwxyz01234567890123456789".

The first message block, is therefore,

$$\begin{aligned} msg[0,\ldots,15] = & \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\ & \texttt{62615A59 66646563 6A696867 6E6D6C6B 7271706F 76757473} \\ & \texttt{7A797877 33323130 37363534 31303938}_x \end{aligned}$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	62615A59	66646563
8	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130	37363534	31303938
16	BCA45940	C167E9D5	DE8485F7	23877E21	DE0D2EE5	B7B71F90	EE756473	CB337449
24	12E7B97F	B166C976	76FCE6D9	55356227	67ED5FC7	2A2F3873	FD062ACA	0E0FA5E9
32	968B6133	3C61C31F	D08B201E	B50C1F12	E26CEDFA	673C3F8E	5B797B61	295F99B3
40	75DB86F1	EA1FB217	5FA37F6A	20EEE4D6	8DF2EDD0	758C4719	DDE8CE1C	83FD4839
48	04FBFE96	A3F15703	7747EC8F	ECA30243	170D7D00	D7AB20E0	4C32D99B	ADF884E0
56	D484DAA8	9E08443D	DCB7BFD3	F36567B9	61B2E83D	7A4A1CD7	A3353EF6	7C646595
64	7EB1E241	00C469F5	0B23891A	9212E002	17C914F5	DC88A9FA	DE203999	BA319015
72	080C7352	40287DA4	66862FC6	FB6914EB	219A9599	1CCC3311	585C2A1D	5DFEF00C
80	EF0E97E6	743BCBEA	DE2AB9A2	54B0D357	BCBB5777	CB8B89AA	6CFF046C	B3BC2838
88	982FF5D2	7B929D1B	20B9D70F	73322680	0746096C	D169852B	0C6449F3	549358D5
96	3E6712CD	785F8219	8AB9E177	6AD7C19A	C4E4D56E	413268DD	0628C5F6	7758FD56
104	D91D9D0F	7DBA58ED	57E12A59	AA2FBB8F	7AFC5181	8688AF72	A64BF27C	2E6F0954
112	26D99711	757E3184	D098571D	5BF34CC5	AC5132CA	D7016AC5	9D3287B1	11E9EC1D
120	C616602E	55C767C9	88B81D1D	EA6869B5	8D208108	6F80925C	48695EA7	BB811ECD
128	4959054D	3D176F23	6B1949D0	12AA4988	91465DE9	BC182315	8F98CE39	80AFB1F4
136	7A0E433B	DA5FA9F0	0817ACE9	90662A8E	577F28F8	67973EB5	D80F7429	ECFE3635

All values are given in hexadecimal.

The value of IV_{224} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	D617833B 68EA6C8F FF3DF700 E5B807EF	6FDB4E75 F966482E 3B40F9B2 755891B2
	Input to $F^3(\cdot)$:	6FDB4E75 F966482E 3B40F9B2 755891B2
	Output from $F^3(\cdot)$:	8ABAE961 D52ADA8E 5BE903F3 04581C9E
1	6FDB4E75 F966482E 3B40F9B2 755891B2	5CAD6A5A BDC0B601 A4D4F4F3 E1E01B71
	Input to $F^3(\cdot)$:	5CAD6A5A BDC0B601 A4D4F4F3 E1E01B71
	Output from $F^3(\cdot)$:	7B2499BE F29D9F73 64AE95BA 9AC7AB76
2	5CAD6A5A BDC0B601 A4D4F4F3 E1E01B71	14FFD7CB 0BFBD75D 5FEE6C08 EF9F3AC4
	Input to $F^3(\cdot)$:	14FFD7CB 0BFBD75D 5FEE6C08 EF9F3AC4
	Output from $F^3(\cdot)$:	35F58D0F 44EEE26D 81B09D48 2A3E911B
3	14FFD7CB 0BFBD75D 5FEE6C08 EF9F3AC4	6958E755 F92E546C 256469BB CBDE8A6A
	Input to $F^3(\cdot)$:	6958E755 F92E546C 256469BB CBDE8A6A
	Output from $F^3(\cdot)$:	85AB5D6A 777FA8A1 442F2400 576040E1
4	6958E755 F92E546C 256469BB CBDE8A6A	91548AA1 7C847FFC 1BC14808 B8FF7A25
	Input to $F^3(\cdot)$:	91548AA1 7C847FFC 1BC14808 B8FF7A25
	Output from $F^3(\cdot)$:	558DCCB8 0F8E397C 02F624F3 B26106C0
5	91548AA1 7C847FFC 1BC14808 B8FF7A25	3CD52BED F6A06D10 27924D48 79BF8CAA
	Input to $F^3(\cdot)$:	3CD52BED F6A06D10 27924D48 79BF8CAA
	Output from $F^3(\cdot)$:	D57E0926 06F2F294 1415171A 1D5EEF96
6	3CD52BED F6A06D10 27924D48 79BF8CAA	442A8387 7A768D68 0FD45F12 A5A195B3
	Input to $F^3(\cdot)$:	442A8387 7A768D68 0FD45F12 A5A195B3
	Output from $F^3(\cdot)$:	B71C74EF B42DA76F 9448DED2 EC88E6A9
7	442A8387 7A768D68 0FD45F12 A5A195B3	8BC95F02 428DCA7F B3DA939A 95376A03
	Input to $F^3(\cdot)$:	8BC95F02 428DCA7F B3DA939A 95376A03
	Output from $F^3(\cdot)$:	BBEAECC3 EE4EB629 FC4CF294 8924B71E
8	8BC95F02 428DCA7F B3DA939A 95376A03	FFC06F44 94383B41 F398AD86 2C8522AD
	Input to $F^3(\cdot)$:	FFC06F44 94383B41 F398AD86 2C8522AD
	Output from $F^3(\cdot)$:	C05E1176 39A34703 E21CC793 80718993
9	FFC06F44 94383B41 F398AD86 2C8522AD	4B974E74 7B2E8D7C 51C65409 1546E390
	Input to $F^3(\cdot)$:	4B974E74 7B2E8D7C 51C65409 1546E390
	Output from $F^3(\cdot)$:	EE93CBB9 6668FA6C 7F7C6B89 A1EC2FD7
10	4B974E74 7B2E8D7C 51C65409 1546E390	1153A4FD F250C12D 8CE4C60F 8D690D7A
	Input to $F^3(\cdot)$:	1153A4FD F250C12D 8CE4C60F 8D690D7A
	Output from $F^3(\cdot)$:	F1C8168B 880B1E1A 6C4B1106 5B0A1A68
11	1153A4FD F250C12D 8CE4C60F 8D690D7A	BA5F58FF F3259366 3D8D450F 4E4CF9F8
	Input to $F^3(\cdot)$:	BA5F58FF F3259366 3D8D450F 4E4CF9F8
	Output from $F^3(\cdot)$:	9A17AC0B 9AA54590 8AC876AD 56C5F011
12	BA5F58FF F3259366 3D8D450F 4E4CF9F8	8B4408F6 68F584BD 062CB0A2 DBACFD6B

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$\begin{aligned} h_0 = &\texttt{D617833B 68EA6C8F FF3DF700 E5B807EF 6FDB4E75 F966482E 3B40F9B2 755891B2} \\ &\oplus \\ &\texttt{BA5F58FF F3259366 3D8D450F 4E4CF9F8 8B4408F6 68F584BD 062CB0A2 DBACFD6B} \\ &= \\ h_1 = &\texttt{6C48DBC4 9BCFFFE9 C2B0B20F ABF4FE17 E49F4683 9193CC93 3D6C4910 AEF46CD9} \end{aligned}$$

The second message block after padding is:

$$\begin{aligned} msg[0,\ldots,15] = &\texttt{35343332 39383736 00000080 00000000 00000000 00000000} \\ &\texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\ &\texttt{00000000 02400000 00000000 00e00000}_x \end{aligned}$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	35343332	39383736	00000080	00000000	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	02400000	00000000	00E00000
16	C152D030	A7BE920F	B92B18A6	3EBEF203	A231B353	C4DDF16C	DA487BC5	5DDD9160
24	C152D030	A7BE920F	B92B18A6	3EBEF203	8A25A76F	59C273EE	DA487BC5	E309FB0A
32	9890A3DE	7DF6E9CA	5A22E3AC	A62E51DD	DFC75A99	9EFF12C0	7C662A18	821ACBF9
40	5FADC2F0	DBD8B817	3B31D35F	611330F3	51FD1F78	62F3A0B1	BB5B4B36	B2F4E472
48	D64AED1A	2845A7BB	DC0C9B1D	28C49A72	64E8C2EC	F169A7F8	CF652B66	49DB7311
56	D7443BFC	7CEE7966	0DC0C330	04710646	069BA52A	61F2F4B7	7A6174D7	59346710
64	B7B819AD	5224D36C	8538FC0D	9F7C83DF	36CC1180	74515BF5	5019A8B9	7F176291
72	A3156009	2CF7D1DF	72D7A1A1	A764664F	2A6C74F5	13255516	DD051298	735813E5
80	5AD8142F	92A591D0	757B0B51	4EC84EA4	65D1517D	5B761C80	78221AE2	6FA89DA1
88	A7E84BD1	50451DF4	27BC3505	75A00CF6	7B229697	A2BA40DA	68CAC3AF	F5A22F9D
96	F86254F5	FA6F527F	80D924CC	B6AA1A51	9FBE0302	DBAF384C	CE8800B3	F0169EA3
104	7C47739D	9ECD1D47	D7AAABA6	09E77F6B	E5EF8BD0	7510EB7C	612DBCC4	104DA44D
112	A4B3FDE7	E92CA703	5367852A	96CC49A9	E6F30F16	7F8DE6EB	D3C81ED6	D94A33D1
120	9388F5C9	EB9A82B9	AA418099	38E2DBC0	F815B044	3DFFC46D	AA1278D2	52F83CC2
128	994C398A	433EDFD1	019FB9E8	0F807023	A5CDD0C7	7E125F03	DC486EF5	7C87E316
136	ED9AAACA	37D2EC4C	D6C6638F	D578710A	CFC75C08	EB39A7E2	7F6A09D8	9D3F60CA

All values are given in hexadecimal.

The value of h_1 is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	6C48DBC4 9BCFFFE9 C2B0B20F ABF4FE17	E49F4683 9193CC93 3D6C4910 AEF46CD9
	Input to $F^3(\cdot)$:	E49F4683 9193CC93 3D6C4910 AEF46CD9
	Output from $F^3(\cdot)$:	ECB230E9 CB47AE3C 4041BA2F E32E3873
1	E49F4683 9193CC93 3D6C4910 AEF46CD9	80FAEB2D 508851D5 82F10820 48DAC664
	Input to $F^3(\cdot)$:	80FAEB2D 508851D5 82F10820 48DAC664
	Output from $F^3(\cdot)$:	A60089DB 970E205C 3B8212F7 DC5A0D42
2	80FAEB2D 508851D5 82F10820 48DAC664	429FCF58 069DECCF 06EE5BE7 72AE619B
	Input to $F^3(\cdot)$:	429FCF58 069DECCF 06EE5BE7 72AE619B
	Output from $F^3(\cdot)$:	9448705D 8E614B8D 42F7E86E BAA05FB2
3	429FCF58 069DECCF 06EE5BE7 72AE619B	14B29B70 DEE91A58 C006E04E F27A99D6
	Input to $F^3(\cdot)$:	14B29B70 DEE91A58 C006E04E F27A99D6
	Output from $F^3(\cdot)$:	0F36ADFD 1AA5E4B8 8271472C EB1CEF84
4	14B29B70 DEE91A58 C006E04E F27A99D6	4DA962A5 1C380877 849F1CCB 99B28E1F
	Input to $F^3(\cdot)$:	4DA962A5 1C380877 849F1CCB 99B28E1F
	Output from $F^3(\cdot)$:	3F9B938D 5074CB65 BD7B23DE DEDE4EEF
5	4DA962A5 1C380877 849F1CCB 99B28E1F	2B2908FD 8E9DD13D 7D7DC390 2CA4D739
	Input to $F^3(\cdot)$:	2B2908FD 8E9DD13D 7D7DC390 2CA4D739
	Output from $F^3(\cdot)$:	1DCBAA9F 69DCA171 C0859DF1 181D1708
6	2B2908FD 8E9DD13D 7D7DC390 2CA4D739	5062C83A 75E4A906 441A813A 81AF9917
	Input to $F^3(\cdot)$:	5062C83A 75E4A906 441A813A 81AF9917
	Output from $F^3(\cdot)$:	209CF738 A306567F BF27FDF0 3E292884
7	5062C83A 75E4A906 441A813A 81AF9917	0BB5FFC5 2D9B8742 C25A3E60 128DFFBD
	Input to $F^3(\cdot)$:	0BB5FFC5 2D9B8742 C25A3E60 128DFFBD
	Output from $F^3(\cdot)$:	AE41E91A 32626ADE 167F681A 104FD43E
8	0BB5FFC5 2D9B8742 C25A3E60 128DFFBD	FE232120 4786C3D8 5265E920 91E04D29
	Input to $F^3(\cdot)$:	FE232120 4786C3D8 5265E920 91E04D29
	Output from $F^3(\cdot)$:	92CE69F3 C5823826 EE9A6013 DD00AAEC
9	FE232120 4786C3D8 5265E920 91E04D29	997B9636 E819BF64 2CC05E73 CF8D5551
	Input to $F^3(\cdot)$:	997B9636 E819BF64 2CC05E73 CF8D5551
	Output from $F^3(\cdot)$:	50E36D70 EFAD837B DCE913B5 55B31795
10	997B9636 E819BF64 2CC05E73 CF8D5551	AEC04C50 A82B40A3 8E8CFA95 C4535ABC
	Input to $F^3(\cdot)$:	AEC04C50 A82B40A3 8E8CFA95 C4535ABC
	Output from $F^3(\cdot)$:	7C8A6DC4 8B08CAE4 E6443AC8 662EEA13
11	AEC04C50 A82B40A3 8E8CFA95 C4535ABC	E5F1FBF2 63117580 CA8464BB A9A3BF42
	Input to $F^3(\cdot)$:	E5F1FBF2 63117580 CA8464BB A9A3BF42
	Output from $F^3(\cdot)$:	2CFE6580 EA96B27F 02C4C114 ACF0C144
12	E5F1FBF2 63117580 CA8464BB A9A3BF42	823E29D0 42BDF2DC 8C483B81 68A39BF8

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$h_1 = \texttt{6C48DBC4 9BCFFFE9 C2B0B20F ABF4FE17 E49F4683 9193CC93 3D6C4910 AEF46CD9} \oplus \texttt{E5F1FBF2 63117580 CA8464BB A9A3BF42 823E29D0 42BDF2DC 8C483B81 68A39BF8}$$
$$=$$
$$h_2 = \texttt{89B92036 F8DE8A69 0834D6B4 02574155 66A16F53 D32E3E4F B1247291 C657F721}$$

This value is truncated to the digest:

$$89B92036\ F8DE8A69\ 0834D6B4\ 02574155\ 66A16F53\ D32E3E4F\ B1247291_x.$$

B.2 Digests of 256 Bits

B.2.1 One-Block Message We outline here the internal state of the computation of computing the 256-bit digest of the message M = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

The first step is the padding of the message, yielding the array of 32-bit words:

$$\begin{aligned} msg[0,\ldots,15] = {} & \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\ & \texttt{00805A59 00000000 00000000 00000000 00000000 00000000} \\ & \texttt{00000000 00D00000 00000000 01000000}_x \end{aligned}$$

The message expansion takes this message and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	00805A59	00000000
8	00000000	00000000	00000000	00000000	00000000	00D00000	00000000	01000000
16	71D0FB0E	116460D2	1B25B312	DE6D4956	3400A604	CF0273BC	4B212655	BFD0372A
24	2F1BBD1F	D41968A7	503A3D4E	A4CB2C31	3400A604	CF0273BC	A15453CA	F4A20E13
32	BED288B2	B0303318	EF87BD01	60BFC1E4	8430951C	2085CEBD	2B9EE7B1	3BE0A236
40	0F9E73A2	FF878F16	6BDA9F78	AB555F93	CB872912	A4D8ECC4	0A010C59	3F252701
48	09844BF5	7033C9B2	5666E9BA	B1678AA6	928AE63D	47D2E40F	DCEF64EC	1C739C81
56	6112CBBB	6A6D5C28	7454F413	A4427AC4	4088616F	EF0EA80B	22D093CF	070EC101
64	E68AE3FE	52E35A7D	516828BB	57ED6958	C069BC40	16BACCB4	8B020DB4	DC1A20C1
72	77A8070F	E16F519C	A84ED4D2	D3EA7DCB	A1E730F3	47407CD9	F13AEE04	A6E9F1F2
80	90C5AA60	D5527899	E9CAC656	FA078893	CF467B86	FF27A706	5EFC4483	3EEFB8C7
88	CB0BF084	34751A73	D5881BAC	B806DFEE	4F402966	ED6287AC	EE049C00	FE81655C
96	7DA72DCC	3B56E499	174BA30A	87A0A55F	F4109F1F	E86C040C	D95CE1DC	CAFF27D8
104	2367F488	ED29FBAF	1F773C74	9B612B66	A269D2C9	F215BBD8	7565B766	5CE8B795
112	BB80CAB9	B0FB8EFD	D57D22BB	037BB5BB	AC7F5035	C2E8A5E8	C879C335	5220131C
120	AC99D6A6	FF7456B2	89B767AA	28916735	7C7C98EE	8CFF4246	3E1890B5	748F601C
128	377F88FF	8EE31E48	A1F242A7	34043D44	229C4E7D	631AE74F	FC7DFE71	70BC5D61
136	CF8331E9	0309A8C3	F90B3ACB	E71256DC	7F75302D	75F4788D	D90AC669	0BFA5031

All values are given in hexadecimal.

The value of IV_{256} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	AE9F3281 5F867848 C988766 D00B409D	31C1F23F 5361AAB9 FB5E1BF6 889EE275
	Input to $F^3(\cdot)$:	31C1F23F 5361AAB9 FB5E1BF6 889EE275
	Output from $F^3(\cdot)$:	5FFEBF42 A0E5D5B3 A93A3E9B C148C2FE
1	31C1F23F 5361AAB9 FB5E1BF6 889EE275	F1618DC3 FF63ADFB A5A2B9FD 11438263
	Input to $F^3(\cdot)$:	F1618DC3 FF63ADFB A5A2B9FD 11438263
	Output from $F^3(\cdot)$:	8D06280D A9D88C2B F5018625 0F2370B7
2	F1618DC3 FF63ADFB A5A2B9FD 11438263	BCC7DA32 FAB92692 0E5F9DD3 87BD92C2
	Input to $F^3(\cdot)$:	BCC7DA32 FAB92692 0E5F9DD3 87BD92C2
	Output from $F^3(\cdot)$:	DD618F4C C623238D 90A52495 696E69C4
3	BCC7DA32 FAB92692 0E5F9DD3 87BD92C2	2C00028F 39408E76 35079D68 782DEBA7
	Input to $F^3(\cdot)$:	2C00028F 39408E76 35079D68 782DEBA7
	Output from $F^3(\cdot)$:	B2CB9B52 85B4BCA4 7C2A89D2 C420D232
4	2C00028F 39408E76 35079D68 782DEBA7	0E0C4160 7F0D9A36 72751401 439D40F0
	Input to $F^3(\cdot)$:	0E0C4160 7F0D9A36 72751401 439D40F0
	Output from $F^3(\cdot)$:	1A4ABFAF 32056210 4C89E60F 0DD7AFEE
5	0E0C4160 7F0D9A36 72751401 439D40F0	364ABD20 0B45EC66 798E7B67 75FA4449
	Input to $F^3(\cdot)$:	364ABD20 0B45EC66 798E7B67 75FA4449
	Output from $F^3(\cdot)$:	FBEDF4D1 B3C02AD3 3D9169AD E2F56A97
6	364ABD20 0B45EC66 798E7B67 75FA4449	F5E1B5B1 CCCDB0E5 4FE47DAC A1682A67
	Input to $F^3(\cdot)$:	F5E1B5B1 CCCDB0E5 4FE47DAC A1682A67
	Output from $F^3(\cdot)$:	F67B3776 86EE3612 FADF6C7A 04202C7C
7	F5E1B5B1 CCCDB0E5 4FE47DAC A1682A67	C0318A56 8DABDA74 8351171D 71DA6835
	Input to $F^3(\cdot)$:	C0318A56 8DABDA74 8351171D 71DA6835
	Output from $F^3(\cdot)$:	BDC3A025 955217D0 A8EDA31E D6071CF2
8	C0318A56 8DABDA74 8351171D 71DA6835	48221594 599FA735 E709DEB2 776F3695
	Input to $F^3(\cdot)$:	48221594 599FA735 E709DEB2 776F3695
	Output from $F^3(\cdot)$:	D1349EC4 66C44275 47664B66 4361EA1D
9	48221594 599FA735 E709DEB2 776F3695	11051492 EB6F9801 C4375C7B 32BB8228
	Input to $F^3(\cdot)$:	11051492 EB6F9801 C4375C7B 32BB8228
	Output from $F^3(\cdot)$:	6D9001B6 30C34EB7 8FF6894B 0471572D
10	11051492 EB6F9801 C4375C7B 32BB8228	25B21422 695CE982 68FF57F9 731E61B8
	Input to $F^3(\cdot)$:	25B21422 695CE982 68FF57F9 731E61B8
	Output from $F^3(\cdot)$:	57281A0C C7D4119D 3C3130F1 F548E7E3
11	25B21422 695CE982 68FF57F9 731E61B8	462D0E9E 2CBB899C F8066C8A C7F365CB
	Input to $F^3(\cdot)$:	462D0E9E 2CBB899C F8066C8A C7F365CB
	Output from $F^3(\cdot)$:	DE01CFC5 469F9CDF E473C074 2BC104A7
12	462D0E9E 2CBB899C F8066C8A C7F365CB	FBB3DBE7 2FC3755D 8C8C978D 58DF651F

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$\begin{aligned} h_0 = {} & \texttt{AE9F3281 5F867848 C988766 D00B409D } \ \texttt{31C1F23F 5361AAB9 FB5E1BF6 889EE275} \\ & \oplus \\ & \texttt{462D0E9E 2CBB899C F8066C8A C7F365CB } \ \texttt{FBB3DBE7 2FC3755D 8C8C978D 58DF651F} \\ & = \\ h_1 = {} & \texttt{E8B23C1F 733DF1D4 F49EEBEC 17F82556 } \ \texttt{CA7229D8 7CA2DFE4 77D28C7B D041876A} \end{aligned}$$

This value is the digest:

$E8B23C1F\ 733DF1D4\ F49EEBEC\ 17F82556\ CA7229D8\ 7CA2DFE4\ 77D28C7B\ D041876A_x$.

B.2.2 Two-Block Message For the two-block message, we picked the message

M = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq
rstuvwxyz01234567890123456789".

The first message block, is therefore,

$$\begin{aligned} msg[0,\ldots,15] = {} & \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\ & \texttt{62615A59 66646563 6A696867 6E6D6C6B 7271706F 76757473} \\ & \texttt{7A797877 33323130 37363534 31303938}_x \end{aligned}$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	62615A59	66646563
8	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130	37363534	31303938
16	BCA45940	C167E9D5	DE8485F7	23877E21	DE0D2EE5	B7B71F90	EE756473	CB337449
24	12E7B97F	B166C976	76FCE6D9	55356227	67ED5FC7	2A2F3873	FD062ACA	0E0FA5E9
32	968B6133	3C61C31F	D08B201E	B50C1F12	E26CEDFA	673C3F8E	5B797B61	295F99B3
40	75DB86F1	EA1FB217	5FA37F6A	20EEE4D6	8DF2EDD0	758C4719	DDE8CE1C	83FD4839
48	04FBFE96	A3F15703	7747EC8F	ECA30243	170D7D00	D7AB20E0	4C32D99B	ADF884E0
56	D484DAA8	9E08443D	DCB7BFD3	F36567B9	61B2E83D	7A4A1CD7	A3353EF6	7C646595
64	7EB1E241	00C469F5	0B23891A	9212E002	17C914F5	DC88A9FA	DE203999	BA319015
72	080C7352	40287DA4	66862FC6	FB6914EB	219A9599	1CCC3311	585C2A1D	5DFEF00C
80	EF0E97E6	743BCBEA	DE2AB9A2	54B0D357	BCBB5777	CB8B89AA	6CFF046C	B3BC2838
88	982FF5D2	7B929D1B	20B9D70F	73322680	0746096C	D169852B	0C6449F3	549358D5
96	3E6712CD	785F8219	8AB9E177	6AD7C19A	C4E4D56E	413268DD	0628C5F6	7758FD56
104	D91D9D0F	7DBA58ED	57E12A59	AA2FBB8F	7AFC5181	8688AF72	A64BF27C	2E6F0954
112	26D99711	757E3184	D098571D	5BF34CC5	AC5132CA	D7016AC5	9D3287B1	11E9EC1D
120	C616602E	55C767C9	88B81D1D	EA6869B5	8D208108	6F80925C	48695EA7	BB811ECD
128	4959054D	3D176F23	6B1949D0	12AA4988	91465DE9	BC182315	8F98CE39	80AFB1F4
136	7A0E433B	DA5FA9F0	0817ACE9	90662A8E	577F28F8	67973EB5	D80F7429	ECFE3635

All values are given in hexadecimal.

The value of IV_{256} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	AE9F3281 5F867848 0C988766 D00B409D	31C1F23F 5361AAB9 FB5E1BF6 889EE275
	Input to $F^3(\cdot)$: 31C1F23F 5361AAB9 FB5E1BF6 889EE275	
	Output from $F^3(\cdot)$: 055F6C94 53F8F887 DDB280E8 605CE712	
1	31C1F23F 5361AAB9 FB5E1BF6 889EE275	ABC05E15 0C7E80CF D12A078E B057A78F
	Input to $F^3(\cdot)$: ABC05E15 0C7E80CF D12A078E B057A78F	
	Output from $F^3(\cdot)$: 2E0094D4 0440ED7F F0A48A3E 57AC826A	
2	ABC05E15 0C7E80CF D12A078E B057A78F	1FC166EB 572147C6 0BFA91C8 DF32601F
	Input to $F^3(\cdot)$: 1FC166EB 572147C6 0BFA91C8 DF32601F	
	Output from $F^3(\cdot)$: 3CD4368F CC62C3BE 9A411360 7D09BD17	
3	1FC166EB 572147C6 0BFA91C8 DF32601F	9714689A C01C4371 4B6B14EE CD5E1A98
	Input to $F^3(\cdot)$: 9714689A C01C4371 4B6B14EE CD5E1A98	
	Output from $F^3(\cdot)$: D131B2DA DEEE14C9 EB3CB85B 3563B415	
4	9714689A C01C4371 4B6B14EE CD5E1A98	CEF0D431 89CF530F E0C62993 EA51D40A
	Input to $F^3(\cdot)$: CEF0D431 89CF530F E0C62993 EA51D40A	
	Output from $F^3(\cdot)$: 9BE92C00 86046A2F FF8DA389 386B5B6E	
5	CEF0D431 89CF530F E0C62993 EA51D40A	0CFD449A 4618295E B4E6B767 F53541F6
	Input to $F^3(\cdot)$: 0CFD449A 4618295E B4E6B767 F53541F6	
	Output from $F^3(\cdot)$: F9AAA05B 90F52485 75516A8A A8DF8284	
6	0CFD449A 4618295E B4E6B767 F53541F6	375A746A 193A778A 95974319 428E568E
	Input to $F^3(\cdot)$: 375A746A 193A778A 95974319 428E568E	
	Output from $F^3(\cdot)$: 9A037C47 DDBF86FF 320602AC 25740751	
7	375A746A 193A778A 95974319 428E568E	96FE38DD 9BA7AFA1 86E0B5CB D04146A7
	Input to $F^3(\cdot)$: 96FE38DD 9BA7AFA1 86E0B5CB D04146A7	
	Output from $F^3(\cdot)$: BB1F1BD7 92A4C5B6 484B6424 42D45DD9	
8	96FE38DD 9BA7AFA1 86E0B5CB D04146A7	8C456FBD 8B9EB23C DDDC273D 005A0B57
	Input to $F^3(\cdot)$: 8C456FBD 8B9EB23C DDDC273D 005A0B57	
	Output from $F^3(\cdot)$: 966CA73F A1C5D462 D3BA0395 B217E658	
9	8C456FBD 8B9EB23C DDDC273D 005A0B57	00929FE2 3A627BC3 555AB65E 6256A0FF
	Input to $F^3(\cdot)$: 00929FE2 3A627BC3 555AB65E 6256A0FF	
	Output from $F^3(\cdot)$: 40F86C89 E5C1A0BD BA2714BD 864AB268	
10	00929FE2 3A627BC3 555AB65E 6256A0FF	CCBD0334 6E5F1281 67FB3380 8610B93F
	Input to $F^3(\cdot)$: CCBD0334 6E5F1281 67FB3380 8610B93F	
	Output from $F^3(\cdot)$: 193F0DFB F6C4213E C8953E42 3EE3E074	
11	CCBD0334 6E5F1281 67FB3380 8610B93F	19AD9219 CCA65AFD 9DCF881C 5CB5408B
	Input to $F^3(\cdot)$: 19AD9219 CCA65AFD 9DCF881C 5CB5408B	
	Output from $F^3(\cdot)$: D5E74F99 046F580F 94E18E01 93C6C196	
12	19AD9219 CCA65AFD 9DCF881C 5CB5408B	195A4CAD 6A304A8E F31ABD81 15D678A9

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$
\begin{aligned}
h_0 = {} & \texttt{AE9F3281 5F867848 0C988766 D00B409D} \;\; \texttt{31C1F23F 5361AAB9 FB5E1BF6 889EE275} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \oplus \\
& \texttt{19AD9219 CCA65AFD 9DCF881C 5CB5408B} \;\; \texttt{195A4CAD 6A304A8E F31ABD81 15D678A9} \\
& \qquad\qquad\qquad\qquad\qquad\qquad = \\
h_1 = {} & \texttt{B732A098 932022B5 91570F7A 8CBE0016} \;\; \texttt{289BBE92 3951E037 0844A677 9D489ADC}
\end{aligned}
$$

The second message block after padding is:

$$
\begin{aligned}
msg[0,\ldots,15] = {} & \texttt{35343332 39383736 00000080 00000000 00000000 00000 000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 02400000 00000000 01000000}_x
\end{aligned}
$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	35343332	39383736	00000080	00000000	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	02400000	00000000	01000000
16	C152D030	A7BE920F	B92B18A6	3F5EF203	A231B353	C4DDF16C	DA487BC5	5C3D9160
24	C152D030	A7BE920F	B92B18A6	3F5EF203	8A25A76F	C4DDF16C	E45764E4	E2E9FB0A
32	058F215C	43E9F6EB	5BC2E3AC	3AD1D35F	E1D845B8	9F1F12C0	E099A89A	BDE5D4D8
40	5E4DC2F0	47273A95	04CECC7E	611330F3	CD029DFA	C0133D12	85445417	2FEB66F0
48	BAB2B003	E4A3FB53	D8758B84	05407BC7	1A1AA10B	99E3CAB3	FE35CAC9	161F7F14
56	4F370D7C	EEAE71B2	766FD37D	AF146A61	13C3BDDB	6CDBF32E	120DF4E9	E5BF65E3
64	D669432D	F6AE0FBA	3DCAEE67	D32938EA	ECB4AEB1	A42924D4	2D1CF223	FAABD1A5
72	EB1E29A8	C3B28391	8CC402D8	440A43C9	D0713E4A	E01FF1F6	5607B720	35CE5BA9
80	170A7554	B4B340D3	79975014	5A148B71	CCEA9FA6	15D0F1B6	BFC7F694	A60DE318
88	6C10C44D	500F3BC1	9276FFE5	9CCEAFD9	0EFDF33E	5F3CB8B4	4DF25133	5D16389D
96	4836CDE0	F94111E0	24816889	12224691	35AB8E46	3151993F	ADE5B005	93A66D5E
104	5D415D72	FDEA8BC4	01D092BB	C18FF2AB	F31778FA	5EEC2A0F	8C7DA398	AE014067
112	759961C5	C0DAD6A2	2F84820A	5000834E	F6E9E9C6	F159CAD1	AFDE462F	B045F969
120	079D0443	CC0B7952	1616E2D6	97AF0AB1	8916C447	4A3E98D2	6D7BED78	4BBE4973
128	3FA7F917	ADA13BDA	643ACB79	6FA77A59	5B48D21C	956301A8	C0793C76	EB0D2B75
136	92FE05EB	0C724524	FD1BC9A3	05510F5A	85648163	B7255171	682AE222	CEDAC810

All values are given in hexadecimal.

The value of h_1 is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left half (L_i)	Right half (R_i)
0	B732A098 932022B5 91570F7A 8CBE0016	289BBE92 3951E037 844A677 9D489ADC
	Input to $F^3(\cdot)$:	289BBE92 3951E037 844A677 9D489ADC
	Output from $F^3(\cdot)$:	A37B07FA 460AE454 491BB340 DCC7BCB8
1	289BBE92 3951E037 844A677 9D489ADC	1449A762 D52AC6E1 D84CBC3A 5079BCAE
	Input to $F^3(\cdot)$:	1449A762 D52AC6E1 D84CBC3A 5079BCAE
	Output from $F^3(\cdot)$:	BA71932B A4D493BF 27FD8F3A 7F013F32
2	1449A762 D52AC6E1 D84CBC3A 5079BCAE	92EA2DB9 9D857388 2FB9294D E249A5EE
	Input to $F^3(\cdot)$:	92EA2DB9 9D857388 2FB9294D E249A5EE
	Output from $F^3(\cdot)$:	25A36E0A 2D724E0E 56066A7D 6A398F4F
3	92EA2DB9 9D857388 2FB9294D E249A5EE	31EAC968 F85888EF 8E4AD647 3A4033E1
	Input to $F^3(\cdot)$:	31EAC968 F85888EF 8E4AD647 3A4033E1
	Output from $F^3(\cdot)$:	DF3E056E ECEA6C8E 705063A1 217A2503
4	31EAC968 F85888EF 8E4AD647 3A4033E1	4DD428D7 716F1F06 5FE94AEC C33380ED
	Input to $F^3(\cdot)$:	4DD428D7 716F1F06 5FE94AEC C33380ED
	Output from $F^3(\cdot)$:	AF8A583C 8E9B4234 C03A0EC7 7B74907F
5	4DD428D7 716F1F06 5FE94AEC C33380ED	9E609154 76C3CADB 4E70D880 4134A39E
	Input to $F^3(\cdot)$:	9E609154 76C3CADB 4E70D880 4134A39E
	Output from $F^3(\cdot)$:	823E31FF 4B80ABF0 2C655205 64EB9ED4
6	9E609154 76C3CADB 4E70D880 4134A39E	CFEA1928 3AEFB4F6 738C18E9 A7D81E39
	Input to $F^3(\cdot)$:	CFEA1928 3AEFB4F6 738C18E9 A7D81E39
	Output from $F^3(\cdot)$:	601C142D E171B3F3 E0F15FB3 1E6A21FB
7	CFEA1928 3AEFB4F6 738C18E9 A7D81E39	FE7C8579 97B27928 AE818733 5F5E8265
	Input to $F^3(\cdot)$:	FE7C8579 97B27928 AE818733 5F5E8265
	Output from $F^3(\cdot)$:	7175B7BB 293D5D48 7CA996F9 4B5378BB
8	FE7C8579 97B27928 AE818733 5F5E8265	BE9FAE93 13D2E9BE F258E10 EC8B6682
	Input to $F^3(\cdot)$:	BE9FAE93 13D2E9BE F258E10 EC8B6682
	Output from $F^3(\cdot)$:	69E21F98 EEF83863 F6657775 942E6A85
9	BE9FAE93 13D2E9BE F258E10 EC8B6682	979E9AE1 794A414B 58E4F046 CB70E8E0
	Input to $F^3(\cdot)$:	979E9AE1 794A414B 58E4F046 CB70E8E0
	Output from $F^3(\cdot)$:	C5F722A4 F4C91B05 B282C7EB C761A21A
10	979E9AE1 794A414B 58E4F046 CB70E8E0	7B688C37 E71BF2BB BDA749FB 2BEAC498
	Input to $F^3(\cdot)$:	7B688C37 E71BF2BB BDA749FB 2BEAC498
	Output from $F^3(\cdot)$:	E849C052 D06D718E A234B708 36DCFFA5
11	7B688C37 E71BF2BB BDA749FB 2BEAC498	7FD75AB3 A92730C5 FAD0474E FDAC1745
	Input to $F^3(\cdot)$:	7FD75AB3 A92730C5 FAD0474E FDAC1745
	Output from $F^3(\cdot)$:	FF58A9C3 81F1C4DC FF2809B4 7975D770
12	7FD75AB3 A92730C5 FAD0474E FDAC1745	843025F4 66EA3667 428F404F 529F13E8

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$\begin{aligned} h_1 = &\texttt{B732A098 932022B5 91570F7A 8CBE0016} \;\; \texttt{289BBE92 3951E037} \;\; \texttt{844A677 9D489ADC} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \oplus \\ &\texttt{7FD75AB3 A92730C5 FAD0474E FDAC1745} \;\; \texttt{843025F4 66EA3667 428F404F 529F13E8} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \\ h_2 = &\texttt{C8E5FA2B 3A071270 6B874834 71121753} \;\; \texttt{ACAB9B66 5FBBD650 4ACBE638 CFD78934} \end{aligned}$$

The digest is h_2:

$C8E5FA2B\ 3A071270\ 6B874834\ 71121753\ ACAB9B66\ 5FBBD650\ 4ACBE638\ CFD78934_x$.

B.3 Digests of 384 Bits

B.3.1 One-Block Message We outline here the internal state of the computation of computing the 384-bit digest of the message M = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

The first step is the padding of the message, yielding the array of 32-bit words:

$$\begin{aligned} msg[0,\ldots,31] = &\texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\ &\texttt{00805A59 00000000 00000000 00000000 00000000 00000000} \\ &\texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\ &\texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\ &\texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\ &\texttt{00000000 00000000 00000000 00000000 00D00000 00000000} \\ &\texttt{00000000 01800000}_x \end{aligned}$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	00805A59	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	00D00000	00000000	00000000	00000000	01800000
32	C6DD21A7	F255D8E5	E9B2B0C3	13374719	EC42280E	6D0CC7C0	791301B7	631BFE5A
40	8F214B6D	0E6FA4A3	1A7062D4	00789D39	EC42280E	6D0CC7C0	791301B7	631BFE5A
48	8F214B6D	0E6FA4A3	1A7062D4	00789D39	EC42280E	6D0CC7C0	791301B7	631BFE5A
56	8F214B6D	3B49B7B0	1A7062D4	00789D39	EC42280E	B16D7A7D	470C1E96	631BFE5A
64	FD949617	E825BA31	E9CA2DFA	FF756F17	5D2F5273	2A00D956	1A08FFED	9E8F684D
72	6704F15C	E7A58959	E5050DC3	5D57CF4A	C642F158	7704382D	E79C69FA	041F0F06
80	6884C234	EB6AA960	4727AD9E	C63A6C61	9B461023	8A90AE3A	7D0C0EB1	0B9F3C6E
88	644BE20D	7C6E1A2E	DC4A0EB5	9B3E8D1A	66D28634	CC6174CC	4C9322F8	07501C57
96	8FB229A5	1B329E76	DFEB924E	C6F9D7B7	C477FCE1	60C78781	96C019D9	0EAAEACB
104	8AFAD233	4E9B1A86	8AAF8004	1497BD39	1E58AB4B	5FB464EB	2B212778	FCA3A32B
112	D0DA06C7	6AB0650E	614E6C25	0AA9FFB6	F843211C	B327551E	F52E8D6D	B6047C64
120	33D43F12	C05C19C7	646F11F1	832610CA	5E9BA3BF	6854363D	E2379D00	07251C55
128	4FEE3062	7F5D8F87	5CCD8284	98627408	AC23CADC	82F01A81	91E5058C	4144DAA9
136	F5A75DB4	12569802	12CDF40C	B8B477E5	9CA8B1CA	CE516167	6A65FDD1	0904FE9F
144	C28C9EC5	787D9102	D9FA1BC0	96014E7C	3612407B	D942A8CF	FC2A73F2	7488E2A1
152	4BA9AE10	19A60207	F26E5F8D	B53450B1	87D90B70	947E45CF	96BF7FA1	4C8CB245
160	DC6DBE7D	F6609491	4E63168E	A8950CED	5BA84BD1	BA47F27E	FF35405C	A669A9CF
168	768942D8	842CB711	7BD5E515	DFF5EB59	633A7540	C017583C	F55FA147	9FB53A08
176	FE0B4BCA	52B09EFA	59259966	708F9517	197DD3AA	FD027DF3	6DEADDC9	B36AEFDF
184	89B37B8E	FD496B1E	9846D2B3	A513BE71	B389CC75	CBE64CF5	C98544F7	39FA57B5
192	2124D563	6E264622	EB70A8FF	1B1CC098	904E0724	73C2B689	C6CF17E9	874D7CAC
200	18AF04FA	6F5C1FEE	60C9258D	4FBBEC7D	10F8C3C9	06D84FD5	7212DDEB	871A3EF2
208	91575424	3279BB77	169E751B	607756DE	1FA59C7F	8F10A018	EAF0E33B	223DBBFB
216	BBCAC0F9	EBD71E05	F831846D	BAB6220E	3C996C6D	2116AFCE	EBB8FF0C	8230974C
224	AB8ADBD1	C74A724C	B883E8A6	E523C8E9	0D0CA4AD	E8D046F8	B7066EC2	2B93DEA5
232	D9612107	ACEAFBDB	B276EF56	74145DD9	2897846C	FC844091	329D8A63	10CDEF35
240	F45C8FDF	A3A7B67C	217124FC	489ADEBC	A5DBD293	ACBA5D56	1437576B	88E9E722
248	349071CE	2DD980B7	35817712	A1555B1B	D50AAF4E	33434BCE	BD206157	9A23A03E
256	86535B66	F2CB055E	19D6B3BD	302967A7	3E4FEF63	55F027AF	2D25CEFC	ADC085C3
264	2BAA2459	B53C4866	825F88F1	4A5BB2BA	7D67A3C3	D1A18E6D	9F5D0FA0	3B67CB6C
272	4160C7B9	21F83E8D	6B2A9646	35FD7D7F	747A5CFE	33E752F6	2F509C07	C989209B
280	15684F43	46F316F1	007C0A6D	D52F07E5	E6EDFDB8	1C13D7C9	74A941CC	8F4BEF7D
288	5834C62D	3A853C82	41ABF0B2	17DA8A7B	96EF2677	E2317066	9BD2A925	42C3CBED
296	912D1466	8A39633A	30108FE3	AD5C4539	08842A2B	85D16DBB	8228E199	380D94C7
304	74E42D81	1C0A3612	369236EE	A4750037	10F81E56	CCD6D644	272565D9	F6B99DF8
312	4BCCEB59	10A94B29	9559E9A6	ED8A77D0	B5FBB532	6867BEC2	C9D90E2E	119F8342
320	489D8D04	AFDCD524	AC218762	A2213F49	FE8898B5	2BE87E48	8A4D2A67	0A5E46E9
328	3EF1C142	2618E458	9231B0AA	53D4DD8C	236C5463	0F9C47DC	8876A770	06FC5585
336	52FCC9D9	8E3B86B8	6546EB62	87195454	1F64598A	44A07134	21D9305C	A4455421
344	C5F76DE1	75EFA04B	1240BDF2	F2EE2E5A	F15BC406	49BE8E9E	6D9C5A0F	D468EEA3
352	98AEA7FA	071F02FA	52B5DD53	CD2F59B2	D0B965FA	07495EC2	BDCA17DC	ED938A26

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	9AAFB8A6	537417F7	4169667B	FD36EC69	394D0315	AD2DB227	9E0CF198	03C40EB4
368	F0B55F2E	E1C33E21	95063B1A	F71B45FF	C7DAB3EA	61BC08A8	2B7DB2B5	AEF171A1
376	34FE5685	1FD25F81	F39608B4	47C22592	7C7552E0	BF4B038A	47F1D1A0	43F7536E
384	877CF87B	F4890A4E	1577F8C1	B15A0B52	6FF26670	40B88F62	FE3D44B2	6AEF725D
392	6E26B2E8	4603EF36	F0336D29	92C48A19	79F58C77	5310F695	F4E383C5	6DE2BC5C
400	B6B6B018	11F05308	07C2B103	8EEEC988	94CA457F	955F8B6D	469F0EE9	1847C1B9
408	250E058D	1810EE82	7D78C13C	D30860ED	E92AD98D	F9D40D63	5FB61019	66F956E3
416	8238A6CB	FC061929	FB8F8EDD	FB78643C	E8EEFB15	72E7FE5C	3B26BA4F	53694230
424	E730286B	670F0FF2	DD4353E7	1390B97E	85A51AC2	FF1D9E80	70C8FC50	87CCD80D
432	85BCF0E6	C18AF67B	061AE72E	9806C608	0D619C84	DC059423	F80666B7	6B119CA6
440	1E99D1BE	1134B894	11BEDD53	7D7C66FF	0909984B	EA76929F	40EAD987	42A68C7D

All values are given in hexadecimal.

The value of IV_{384} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
0	1E41CEC0 E742F23B 5E195589 DDCFE7A0	827678F1 97AB48F6 5306C06C 64879
	15FE61A9 79FFC139 10426AA1 F255945E	5573B567 B9BDA1CA CEF5447F 1A4A03A7
	Input to left $F^4(\cdot)$:	827678F1 97AB48F6 5306C06C 00064879
	Input to right $F^4(\cdot)$:	5573B567 B9BDA1CA CEF5447F 1A4A03A7
	Output from left $F^4(\cdot)$:	E084AF2C 037FEAF4 ED73DB23 9076BEBB
	Output from right $F^4(\cdot)$:	C2F6BA25 7D83C11B FDA890B4 F129D7BE
1	827678F1 97AB48F6 5306C06C 00064879	D708DB8C 047C0022 EDEAFA15 037C43E0
	5573B567 B9BDA1CA CEF5447F 1A4A03A7	FEC561EC E43D18CF B36A8EAA 4DB9591B
	Input to left $F^4(\cdot)$:	D708DB8C 047C0022 EDEAFA15 037C43E0
	Input to right $F^4(\cdot)$:	FEC561EC E43D18CF B36A8EAA 4DB9591B
	Output from left $F^4(\cdot)$:	30A89F9D AB315BF4 5A507DA5 FBEE011D
	Output from right $F^4(\cdot)$:	DAAD6F3C E41C03C6 88C82E55 1D3660DA
2	D708DB8C 047C0022 EDEAFA15 037C43E0	8FDEDA5B 5DA1A20C 463D6A2A 077C637D
	FEC561EC E43D18CF B36A8EAA 4DB9591B	B2DEE76C 3C9A1302 0956BDC9 FBE84964
	Input to left $F^4(\cdot)$:	8FDEDA5B 5DA1A20C 463D6A2A 077C637D
	Input to right $F^4(\cdot)$:	B2DEE76C 3C9A1302 0956BDC9 FBE84964
	Output from left $F^4(\cdot)$:	34A77FBC 17B44C8C EF9B1A33 2F6EAFA8
	Output from right $F^4(\cdot)$:	CDC7C8BC 50403B94 9520A578 48FD8E12
3	8FDEDA5B 5DA1A20C 463D6A2A 077C637D	3302A950 B47D235B 264A2BD2 0544D709
	B2DEE76C 3C9A1302 0956BDC9 FBE84964	E3AFA430 13C84CAE 0271E026 2C12EC48
	Input to left $F^4(\cdot)$:	3302A950 B47D235B 264A2BD2 0544D709
	Input to right $F^4(\cdot)$:	E3AFA430 13C84CAE 0271E026 2C12EC48
	Output from left $F^4(\cdot)$:	A304AADF 3FB74649 C8EB54A2 6CB52A08
	Output from right $F^4(\cdot)$:	AD9D4D3E D05DF531 8FBB327A 4B3C7C0E
4	3302A950 B47D235B 264A2BD2 0544D709	1F43AA52 ECC7E633 86ED8FB3 B0D4356A
	E3AFA430 13C84CAE 0271E026 2C12EC48	2CDA7084 6216E445 8ED63E88 6BC94975
	Input to left $F^4(\cdot)$:	1F43AA52 ECC7E633 86ED8FB3 B0D4356A
	Input to right $F^4(\cdot)$:	2CDA7084 6216E445 8ED63E88 6BC94975
	Output from left $F^4(\cdot)$:	4C9DE550 E57E1EB1 38968A09 E3F04F81
	Output from right $F^4(\cdot)$:	38FE0AF9 39D07476 6A713C31 363EC593
5	1F43AA52 ECC7E633 86ED8FB3 B0D4356A	DB51AEC9 2A1838D8 6800DC17 1A2C29DB
	2CDA7084 6216E445 8ED63E88 6BC94975	7F9F4C00 51033DEA 1EDCA1DB E6B49888
	Input to left $F^4(\cdot)$:	DB51AEC9 2A1838D8 6800DC17 1A2C29DB
	Input to right $F^4(\cdot)$:	7F9F4C00 51033DEA 1EDCA1DB E6B49888
	Output from left $F^4(\cdot)$:	6A8C583F 4F772508 0A1CDE01 BE47591F
	Output from right $F^4(\cdot)$:	EB70C288 82D297B1 AE0CFF22 DC129E79
6	DB51AEC9 2A1838D8 6800DC17 1A2C29DB	C7AAB20C E0C473F4 20DAC1AA B7DBD70C
	7F9F4C00 51033DEA 1EDCA1DB E6B49888	75CFF26D A3B0C33B 8CF151B2 0E936C75
	Input to left $F^4(\cdot)$:	C7AAB20C E0C473F4 20DAC1AA B7DBD70C
	Input to right $F^4(\cdot)$:	75CFF26D A3B0C33B 8CF151B2 0E936C75
	Output from left $F^4(\cdot)$:	21BD7A21 A54BB877 FBB02FB6 5FE7FCC6
	Output from right $F^4(\cdot)$:	6BDD59C4 D7AA7A32 0E279D5B 565C824E
7	C7AAB20C E0C473F4 20DAC1AA B7DBD70C	144215C4 86A947D8 10FB3C80 B0E81AC6
	75CFF26D A3B0C33B 8CF151B2 0E936C75	FAECD4E8 8F5380AF 93B0F3A1 45CBD51D
	Input to left $F^4(\cdot)$:	144215C4 86A947D8 10FB3C80 B0E81AC6
	Input to right $F^4(\cdot)$:	FAECD4E8 8F5380AF 93B0F3A1 45CBD51D
	Output from left $F^4(\cdot)$:	E3AA4A2D 25B34021 823454A1 A0FCEB26
	Output from right $F^4(\cdot)$:	57D8F526 A231185B 18F64C71 27860A3D

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
8	144215C4 86A947D8 10FB3C80 B0E81AC6	2217074B 0181DB60 94071DC3 29156648
	FAECD4E8 8F5380AF 93B0F3A1 45CBD51D	2400F821 C57733D5 A2EE950B 17273C2A
	Input to left $F^4(\cdot)$:	2217074B 0181DB60 94071DC3 29156648
	Input to right $F^4(\cdot)$:	2400F821 C57733D5 A2EE950B 17273C2A
	Output from left $F^4(\cdot)$:	973F69B6 E60F1D4E 70A474E9 D1912A3E
	Output from right $F^4(\cdot)$:	31509EC5 DD8BBF33 3D3DCD82 0BD12CE3
9	2217074B 0181DB60 94071DC3 29156648	CBBC4A2D 52D83F9C AE8D3E23 4E1AF9FE
	2400F821 C57733D5 A2EE950B 17273C2A	837D7C72 60A65A96 605F4869 617930F8
	Input to left $F^4(\cdot)$:	CBBC4A2D 52D83F9C AE8D3E23 4E1AF9FE
	Input to right $F^4(\cdot)$:	837D7C72 60A65A96 605F4869 617930F8
	Output from left $F^4(\cdot)$:	BAD3D187 52285A60 DD236FF5 4C7E48C8
	Output from right $F^4(\cdot)$:	0B881526 FF23247F A9983685 00E4646F
10	CBBC4A2D 52D83F9C AE8D3E23 4E1AF9FE	2F88ED07 3A5417AA 0B76A38E 17C35845
	837D7C72 60A65A96 605F4869 617930F8	98C4D6CC 53A98100 49247236 656B2E80
	Input to left $F^4(\cdot)$:	2F88ED07 3A5417AA 0B76A38E 17C35845
	Input to right $F^4(\cdot)$:	98C4D6CC 53A98100 49247236 656B2E80
	Output from left $F^4(\cdot)$:	E8BB7F8B E2592289 93B58EE4 1A9FFAC7
	Output from right $F^4(\cdot)$:	FDCDE2D1 7290BF4A 25DA7073 80E517FE
11	2F88ED07 3A5417AA 0B76A38E 17C35845	7EB09EA3 1236E5DC 4585381A E19C2706
	98C4D6CC 53A98100 49247236 656B2E80	230735A6 B0811D15 3D38B0C7 54850339
	Input to left $F^4(\cdot)$:	7EB09EA3 1236E5DC 4585381A E19C2706
	Input to right $F^4(\cdot)$:	230735A6 B0811D15 3D38B0C7 54850339
	Output from left $F^4(\cdot)$:	40DB1251 4ADB0813 4FC2C7C2 9584C349
	Output from right $F^4(\cdot)$:	05C161F0 F5D86412 105BA9EE ED8511CD
12	7EB09EA3 1236E5DC 4585381A E19C2706	9D05B73C A671E512 597FDBD8 88EE3F4D
	230735A6 B0811D15 3D38B0C7 54850339	6F53FF56 708F1FB9 44B4644C 82479B0C
	Input to left $F^4(\cdot)$:	9D05B73C A671E512 597FDBD8 88EE3F4D
	Input to right $F^4(\cdot)$:	6F53FF56 708F1FB9 44B4644C 82479B0C
	Output from left $F^4(\cdot)$:	CE844D79 19CE4BBB B8785F1E 7C759646
	Output from right $F^4(\cdot)$:	E4C6ED64 DB37B816 8048D12C B55B8EF5
13	9D05B73C A671E512 597FDBD8 88EE3F4D	C7C1D8C2 6BB6A503 BD7061EB E1DE8DCC
	6F53FF56 708F1FB9 44B4644C 82479B0C	B034D3DA 0BF8AE67 FDFD6704 9DE9B140
	Input to left $F^4(\cdot)$:	C7C1D8C2 6BB6A503 BD7061EB E1DE8DCC
	Input to right $F^4(\cdot)$:	B034D3DA 0BF8AE67 FDFD6704 9DE9B140
	Output from left $F^4(\cdot)$:	F526C3F4 F233E561 003CCE5A A9799100
	Output from right $F^4(\cdot)$:	A846E4AA A32DB556 EB10F4E5 785A1E87
14	C7C1D8C2 6BB6A503 BD7061EB E1DE8DCC	C7151BFC D3A2AAEF AFA490A9 FA1D858B
	B034D3DA 0BF8AE67 FDFD6704 9DE9B140	682374C8 54420073 59431582 2197AE4D

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$
\begin{aligned}
h_0 = {} & \texttt{1E41CEC0 E742F23B 5E195589 DDCFE7A0 827678F1 97AB48F6 5306C06C 00064879} \\
& \texttt{15FE61A9 79FFC139 10426AA1 F255945E 5573B567 B9BDA1CA CEF5447F 1A4A03A7} \\
& \qquad\qquad\qquad\qquad \oplus \\
& \texttt{C7C1D8C2 6BB6A503 BD7061EB E1DE8DCC C7151BFC D3A2AAEF AFA490A9 FA1D858B} \\
& \texttt{B034D3DA 0BF8AE67 FDFD6704 9DE9B140 682374C8 54420073 59431582 2197AE4D} = \\
h_1 = {} & \texttt{D9801602 8CF45738 E3693462 3C116A6C 4563630D 4409E219 FCA250C5 FA1BCDF2} \\
& \texttt{A5CAB273 72076F5E EDBF0DA5 6FBC251E 3D50C1AF EDFFA1B9 97B651FD 3BDDADEA}
\end{aligned}
$$

This value is truncated to the digest:

$$
D9801602\ 8CF45738\ E3693462\ 3C116A6C\ 4563630D\ 4409E219\ FCA250C5\ FA1BCDF2\\
A5CAB273\ 72076F5E\ EDBF0DA5\ 6FBC251E_x
$$

B.3.2 Two-Block Message For a two block message, we picked the message

$$
\begin{aligned}
M = {} & \text{“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq} \\
& \text{rstuvwxyz01234567890123456789ABCDEFGHIJKLMNOPQR} \\
& \text{STUVWXYZabcdefghijklmnopqrstuvwxyz0123456789012} \\
& \text{3456789ABCDEFGHIJKLMNOP”.}
\end{aligned}
$$

The first message block, is therefore,

$$
\begin{aligned}
msg[0,\ldots,31] = {} & \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\
& \texttt{62615A59 66646563 6A696867 6E6D6C6B 7271706F 76757473} \\
& \texttt{7A797877 33323130 37363534 31303938 35343332 39383736} \\
& \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655} \\
& \texttt{62615A59 66656463 6A696867 6E6D6C6B 7271706F 76757473} \\
& \texttt{7A797877 33323130}_x
\end{aligned}
$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	62615A59	66656463
8	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130	37363534	31303938
16	35343332	39383736	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655
24	62615A59	66656463	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130
32	B4AC5518	8420AC96	93CBC8B4	21857629	D8173EB3	E0EC54DD	A33A2930	C9317C41
40	14FDA929	E63D823B	3BB3AB9A	57376A2F	61F74F91	7D74733E	B0496789	0C0DADE1
48	121C96A1	9C1A2FE5	251D9CE0	05FA495D	127D1C7E	040EDDD5	C1AA737B	3C3CFB41
56	FF9C300F	015692BC	08B0117C	9EBC1752	2129C87D	15B643BE	0C764AE2	1A06A5BB
64	B5FAC7A4	8C90BDEA	0D77DFE6	00ACBE54	CDA17D0D	EC9A1E3F	B93C8C8B	7CCBBBE5
72	986D14C3	EB4A5DDD	3B1F15CE	9A961722	8D6D51AE	C448FFB5	CC82DC6C	9460B922
80	F956CB7C	A7053A2B	BF8B8BC2	889718F3	D635E3CB	C88C01B9	55CACA59	C56A303D
88	58990A24	BEDD197E	8027098F	4889F499	E9A5C9C4	407C89E7	C91C7ADF	429FAF9F
96	C290C4A1	1C44CAC7	8C25492A	ED135D45	5E614C5C	1B43F937	B5FAC820	E2B0E0EC
104	1236EC10	544E3CF9	5D65528E	C5396C94	AD1278A2	AD037A94	1B559062	9CC2DA9A
112	EFC868F6	D0D0CCE2	E556914E	38C74566	A3DBDD9B	724DEA42	A4881FD8	30425639
120	221A806D	958F4033	8E0985F2	535CD680	EAA7C3AB	0B17407A	E5454300	5ECC01C5
128	571F8492	924D4F35	DF799FAA	07B49EEE	55760C26	FE06BA37	EB36C9E5	B5AF647E
136	807BA325	8B37A353	5AD1CC60	904F60B2	5314C295	4635B371	AEFAF41C	1CB979BF
144	64FFCBA5	8A010082	7519F1FC	6BD387F3	E5EE6EEA	DCB71E5E	B8316667	54BD9D9C
152	A81B80EF	E096B1CF	E5DA0201	B6B2B86A	3610DDF5	B326261D	B1F8DE9C	F6D7812A
160	2D7AE307	D1CE2A41	539D27CF	8D0FECAF	FBF9D962	F756E8C9	87E649CE	F1A2D4FF
168	2CEC20B5	E008EA5F	FE0A77FE	B9AA4C0C	73075E94	4F8CEC46	395274F5	6EA24A19
176	F9F5D0D9	C8EF1CA5	2E14107A	C0EC3B1B	6583BFC4	5995C744	E1776851	64F75456
184	F6A7AB62	EEE42DC1	2343861C	2595EEE8	05BAAE4F	12CC851E	CB528240	D5A712DC
192	C39ECEC6	F28DAC5D	7608C927	88B542E0	E9355C7C	3C046A89	52415B12	323C1A39
200	DE618CE8	96002378	76BF351E	509F1070	4F03341D	1DCDB754	0B6E6ECC	B0C3C6F1
208	6FF5F3A1	BE5029BB	7E8B000A	8FEF0F06	784E0890	52FBA988	51B4AEA0	0B02A7F7
216	48F782D9	906F2DCB	ACAC891A	5DDBE678	574107C7	43782BBE	C05025B7	9D509005
224	96682128	903B1F30	13E69CF9	1A597A2E	7773260F	5B8F92C4	3F3D6D81	00E1D2E3
232	052D874B	850A6650	E7B30B59	58CA6893	FCCF38C4	9975335B	5D09032F	A9348C7F
240	B8B31102	F9DE2048	483E6E37	1BF00072	08368413	2D75B1F4	5CC5EC6D	E50DD896
248	DBD32DC6	C8A79EC8	06889E56	B95AA80C	BC09F54C	39410FCC	7C6A5A8F	84488CB4
256	5ECFBFE0	96B38166	AABC34F5	A6508F62	4E3229C3	27E5C84B	BB75E135	5E2E6D03
264	939E062D	2FB652A5	41E3843B	16F84150	DB2AF08F	2200D26E	03276E2C	3AAA8A52
272	970543A7	B83DA473	5EC62F67	C0DAF0FD	2A36567D	2E52DFD8	666F663F	72089B31
280	63EE89B5	9661B1AF	C6526EAB	936CFE71	925B2A94	5F2E69F3	0E62C1BE	E7A60501
288	947501F0	D20FD321	29BB4683	52A6E5D3	B9F25A42	58C097ED	82105797	FF487575
296	FB833044	EF768151	F3A3268A	82BAACBD	846206BE	E55EEFD9	1847A1A3	D17F5391
304	4BAE398A	39095E79	06757645	F2D22A88	21179280	DFF3876F	62F38BF4	D7A5FB1A
312	74840B21	826CBDAD	135E52AF	2C823421	5D9BDEEE	4BB6F490	3C06FB8F	E55B3E41
320	1619BC5D	C151818E	053972A2	0F3D3B3D	F244AED2	64C66C62	674B69D6	E951C928
328	3AD2B1CA	EA4FF3F3	FC9E1DB7	70FE026F	E0A46ADC	8215860F	F116688B	EBADE25B
36	A1E1CA79	C59743CE	768B742A	12764054	A302148F	2EE5EFE4	895E69AF	76443163
344	B11348EF	F4E7C987	012812FB	8F8020AE	737E310A	C2E89D3F	4A42CAEC	544876AE
352	1B5350EA	017B30F2	F7A93EDA	D55109B8	3BA4C530	36DA521C	40879A44	D68F811D

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	30D83B36	5E49C5BE	82198446	564335ED	C170D5CB	D7A96F3D	E993EB11	18E64CBB
368	6E019D79	10F60134	8C7ADA62	4D52A9B7	50A1F453	FB1BE9F5	C3AFBB70	2B1A8564
376	016B70AF	0A07C5C0	7055055B	2A4567A9	6039A01A	F9524BDE	AEA42311	099FAC1F
384	1154952A	712E35A9	DDEC5973	B568A9A2	C2F68EEE	987E710D	4918365B	C7DB1437
392	41F60E9F	83A59CCD	37712DE4	94B5BB03	590EA4C6	9EB15966	2E48FF26	59104224
400	EDA401B4	27872CD0	18CF6161	145C0D71	CE10AD35	D55316D3	9ABFF954	C6BE84D0
408	26EC5C7F	12C8A4A1	6409082A	E455CA9C	B56AB6C9	63EDB28A	681AA7C1	2F73F060
416	97C552FB	CC0B0C00	A0A1EF16	92EF8073	25FF2E36	A4A5DE58	6B247DD4	49A2E9C5
424	29B0D8FD	F9D500F2	5E9201FC	FD254A73	51DF1EED	A4833C44	09A5EE2F	13693F87
432	0DF361FE	374EB37E	A3B195C7	CC9B46DD	DA97759B	46733E5A	F26305CD	041199CF
440	F6123811	A73F474D	C5B30420	3CA39FB2	B7E41E88	97D393BF	A683F5DA	FEE8CC64

All values are given in hexadecimal.

The value of IV_{384} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
0	1E41CEC0 E742F23B 5E195589 DDCFE7A0	827678F1 97AB48F6 5306C06C 00064879
	15FE61A9 79FFC139 10426AA1 F255945E	5573B567 B9BDA1CA CEF5447F 1A4A03A7
	Input to left $F^4(\cdot)$:	827678F1 97AB48F6 5306C06C 00064879
	Input to right $F^4(\cdot)$:	5573B567 B9BDA1CA CEF5447F 1A4A03A7
	Output from left $F^4(\cdot)$:	13ABAED8 116C5011 8F62BE46 F0497814
	Output from right $F^4(\cdot)$:	145B5F57 02D72E44 792C7C00 C5654057
1	827678F1 97AB48F6 5306C06C 00064879	01A53EFE 7B28EF7D 696E16A1 3730D409
	5573B567 B9BDA1CA CEF5447F 1A4A03A7	0DEA6018 F62EA22A D17BEBCF 2D869FB4
	Input to left $F^4(\cdot)$:	01A53EFE 7B28EF7D 696E16A1 3730D409
	Input to right $F^4(\cdot)$:	0DEA6018 F62EA22A D17BEBCF 2D869FB4
	Output from left $F^4(\cdot)$:	5D3855CF 50A51248 17AFBFEC CE0409A0
	Output from right $F^4(\cdot)$:	90E9FE1A 080B70AB 9E76D39C 8D970ACC
2	01A53EFE 7B28EF7D 696E16A1 3730D409	C59A4B7D B1B6D161 508397E3 97DD096B
	0DEA6018 F62EA22A D17BEBCF 2D869FB4	DF4E2D3E C70E5ABE 44A97F80 CE0241D9
	Input to left $F^4(\cdot)$:	C59A4B7D B1B6D161 508397E3 97DD096B
	Input to right $F^4(\cdot)$:	DF4E2D3E C70E5ABE 44A97F80 CE0241D9
	Output from left $F^4(\cdot)$:	2BE4D9DD D6B83855 AB0095D8 FD8D9C99
	Output from right $F^4(\cdot)$:	796ECB59 692D8222 AC8235C3 EE54E46A
3	C59A4B7D B1B6D161 508397E3 97DD096B	7484AB41 9F032008 7DF9DE0C C3D27BDE
	DF4E2D3E C70E5ABE 44A97F80 CE0241D9	2A41E723 AD90D728 C26E8379 CABD4890
	Input to left $F^4(\cdot)$:	7484AB41 9F032008 7DF9DE0C C3D27BDE
	Input to right $F^4(\cdot)$:	2A41E723 AD90D728 C26E8379 CABD4890
	Output from left $F^4(\cdot)$:	0C705A01 C274BD27 A60BBE81 F87129C8
	Output from right $F^4(\cdot)$:	2A23893B FF5461F6 6EA91836 8D69DC9F
4	7484AB41 9F032008 7DF9DE0C C3D27BDE	F56DA405 385A3B48 2A0067B6 436B9D46
	2A41E723 AD90D728 C26E8379 CABD4890	C9EA117C 73C26C46 F6882962 6FAC20A3
	Input to left $F^4(\cdot)$:	F56DA405 385A3B48 2A0067B6 436B9D46
	Input to right $F^4(\cdot)$:	C9EA117C 73C26C46 F6882962 6FAC20A3
	Output from left $F^4(\cdot)$:	BC123A3B 32A8C107 35F76A6F D94CE75D
	Output from right $F^4(\cdot)$:	E2747D27 C574CB73 60ADC215 EBE9F1ED
5	F56DA405 385A3B48 2A0067B6 436B9D46	C8359A04 68E41C5B A2C3416C 2154B97D
	C9EA117C 73C26C46 F6882962 6FAC20A3	C896917A ADABE10F 480EB463 1A9E9C83
	Input to left $F^4(\cdot)$:	C8359A04 68E41C5B A2C3416C 2154B97D
	Input to right $F^4(\cdot)$:	C896917A ADABE10F 480EB463 1A9E9C83
	Output from left $F^4(\cdot)$:	D824A0ED D6D8E004 FFF23833 2776C689
	Output from right $F^4(\cdot)$:	24B5BE93 867DBF54 4C02508C A086EB12
6	C8359A04 68E41C5B A2C3416C 2154B97D	ED5FAFEF F5BFD312 BA8A79EE CF2ACBB1
	C896917A ADABE10F 480EB463 1A9E9C83	2D4904E8 EE82DB4C D5F25F85 641D5BCF
	Input to left $F^4(\cdot)$:	ED5FAFEF F5BFD312 BA8A79EE CF2ACBB1
	Input to right $F^4(\cdot)$:	2D4904E8 EE82DB4C D5F25F85 641D5BCF
	Output from left $F^4(\cdot)$:	B774D3E0 0972AF8A 58B69096 B6F536EA
	Output from right $F^4(\cdot)$:	091E293B 30C841E9 3A9FA34F FF577715
7	ED5FAFEF F5BFD312 BA8A79EE CF2ACBB1	C188B841 9D63A0E6 7291172C E5C9EB96
	2D4904E8 EE82DB4C D5F25F85 641D5BCF	7F4149E4 6196B3D1 FA75D1FA 97A18F97
	Input to left $F^4(\cdot)$:	C188B841 9D63A0E6 7291172C E5C9EB96
	Input to right $F^4(\cdot)$:	7F4149E4 6196B3D1 FA75D1FA 97A18F97
	Output from left $F^4(\cdot)$:	FCC8A0F1 9768A3F2 9BF36D7C BF9FE4B9
	Output from right $F^4(\cdot)$:	A19D9FD5 D9667A49 314C3E74 28B08F0A

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
8	C188B841 9D63A0E6 7291172C E5C9EB96	8CD49B3D 37E4A105 E4BE61F1 4CADD4C5
	7F4149E4 6196B3D1 FA75D1FA 97A18F97	11970F1E 62D770E0 21791492 70B52F08
	Input to left $F^4(\cdot)$:	8CD49B3D 37E4A105 E4BE61F1 4CADD4C5
	Input to right $F^4(\cdot)$:	11970F1E 62D770E0 21791492 70B52F08
	Output from left $F^4(\cdot)$:	92CCEBB1 0FCA1089 4200EA0F 0F8813CF
	Output from right $F^4(\cdot)$:	074C417B 57BCB36E 5A648A71 148357DD
9	8CD49B3D 37E4A105 E4BE61F1 4CADD4C5	780D089F 362A00BF A0115B8B 8322D84A
	11970F1E 62D770E0 21791492 70B52F08	534453F0 92A9B06F 3091FD23 EA41F859
	Input to left $F^4(\cdot)$:	780D089F 362A00BF A0115B8B 8322D84A
	Input to right $F^4(\cdot)$:	534453F0 92A9B06F 3091FD23 EA41F859
	Output from left $F^4(\cdot)$:	43F3CFD8 2A671428 6D96D918 B7D9E268
	Output from right $F^4(\cdot)$:	9CC6E735 ED70FB2D 183ED27A 16FF6AF9
10	780D089F 362A00BF A0115B8B 8322D84A	8D51E82B 8FA78BCD 3947C6E8 664A45F1
	534453F0 92A9B06F 3091FD23 EA41F859	CF2754E5 1D83B52D 8928B8E9 FB7436AD
	Input to left $F^4(\cdot)$:	8D51E82B 8FA78BCD 3947C6E8 664A45F1
	Input to right $F^4(\cdot)$:	CF2754E5 1D83B52D 8928B8E9 FB7436AD
	Output from left $F^4(\cdot)$:	65DE2FD1 4DEFE7F0 F8AACE6C 866E0011
	Output from right $F^4(\cdot)$:	AB152CBD A9BC4B82 4B74E34E 40610CB6
11	8D51E82B 8FA78BCD 3947C6E8 664A45F1	F8517F4D 3B15FBED 7BE51E6D AA20F4EF
	CF2754E5 1D83B52D 8928B8E9 FB7436AD	1DD3274E 7BC5E74F 58BB95E7 054CD85B
	Input to left $F^4(\cdot)$:	F8517F4D 3B15FBED 7BE51E6D AA20F4EF
	Input to right $F^4(\cdot)$:	1DD3274E 7BC5E74F 58BB95E7 054CD85B
	Output from left $F^4(\cdot)$:	2F4028C2 FC87D32A CD278602 F63B0B3D
	Output from right $F^4(\cdot)$:	881D98F8 1A6CC67A 5F812E00 09FD6A9B
12	F8517F4D 3B15FBED 7BE51E6D AA20F4EF	473ACC1D 07EF7357 D6A996E9 F2895C36
	1DD3274E 7BC5E74F 58BB95E7 054CD85B	A211C0E9 732058E7 F46040EA 90714ECC
	Input to left $F^4(\cdot)$:	473ACC1D 07EF7357 D6A996E9 F2895C36
	Input to right $F^4(\cdot)$:	A211C0E9 732058E7 F46040EA 90714ECC
	Output from left $F^4(\cdot)$:	4CAED17B BA3F2C20 8A0F3BDC 2FC601F9
	Output from right $F^4(\cdot)$:	C1F357CD DB9751D3 0156E51C F809C494
13	473ACC1D 07EF7357 D6A996E9 F2895C36	DC207083 A052B69C 59ED70FB FD451CCF
	A211C0E9 732058E7 F46040EA 90714ECC	B4FFAE36 812AD7CD F1EA25B1 85E6F516
	Input to left $F^4(\cdot)$:	DC207083 A052B69C 59ED70FB FD451CCF
	Input to right $F^4(\cdot)$:	B4FFAE36 812AD7CD F1EA25B1 85E6F516
	Output from left $F^4(\cdot)$:	F1538397 A2D29C08 4BD502B9 F54A4659
	Output from right $F^4(\cdot)$:	D51E5E07 07E3E5EC C13BB063 F152B25F
14	DC207083 A052B69C 59ED70FB FD451CCF	770F9EEE 74C3BD0B 355BF089 6123FC93
	B4FFAE36 812AD7CD F1EA25B1 85E6F516	B6694F8A A53DEF5F 9D7C9450 07C31A6F

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$
\begin{aligned}
h_0 = {} & \texttt{1E41CEC0 E742F23B 5E195589 DDCFE7A0 827678F1 97AB48F6 5306C06C 64879} \\
& \texttt{15FE61A9 79FFC139 10426AA1 F255945E 5573B567 B9BDA1CA CEF544 7F 1A4A03A7} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \oplus \\
& \texttt{DC207083 A052B69C 59ED70FB FD451CCF 770F9EEE 74C3BD0B 355BF089 6123FC93} \\
& \texttt{B4FFAE36 812AD7CD F1EA25B1 85E6F516 B6694F8A A53DEF5F 9D7C9450 07C31A6F} \\
& \qquad\qquad\qquad\qquad\qquad\qquad = \\
h_1 = {} & \texttt{C261BE43 471044A7 7F42572 208AFB6F F579E61F E368F5FD 665D30E5 6125B4EA} \\
& \texttt{A101CF9F F8D516F4 E1A84F10 77B36148 E31AFAED 1C804E95 5389D02F 1D8919C8}
\end{aligned}
$$

The second message block after padding is:

$$
\begin{aligned}
\text{msg}[0,\ldots,31] = {} & \texttt{37363534 31303938 35343332 39383736 00000080 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 04800000 00000000 00000000} \\
& \texttt{00000000 01800000}_x
\end{aligned}
$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	37363534	31303938	35343332	39383736	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	04800000	00000000	00000000	00000000	01800000
32	3F5ADC60	AF3EAF3E	1A0D5667	E4490C78	F2975644	CC5DCC5D	796E3504	872A6F1B
40	91F43527	AF3EAF3E	1A0D5667	E4490C78	F2975644	CC5DCC5D	796E3504	872A6F1B
48	91F43527	AF3EAF3E	1A0D5667	E4490C78	F2975644	CC5DCC5D	796E3504	872A6F1B
56	91F43527	46790190	239CC7CF	E4490C78	F2975644	CC5DCC5D	7EE0BB8D	872A6F1B
64	7923DDF0	8CA268F1	FE445A1F	16DE5A3C	3ECA9A19	B2BD77D0	FE445A1F	FE09B2EB
72	1D565DD6	517AF521	0CD30C5B	DA839661	402A2194	32199642	876787EF	9A7C32CD
80	C08EC006	A3EDA365	C08EC006	A4632DEC	C08EC006	4B3A4BB2	E31207C9	47A4AF1D
88	32199642	86F7C196	87FFEA23	24C7CC7E	B9AD1DF6	2F4FCB94	39441490	B533F959
96	6EAA8AFC	05370CC7	5840A4A1	D9C334D1	CE26AF3A	A32D4D48	9CD7C16D	7FB5BB11
104	BB662253	DA39DB10	25F1BBB1	48480807	6813A4D4	153E499C	35ECA7B2	977627BE
112	E037ADD5	9BF1B092	E9E652A4	737DEE03	CBE99475	EF90DE3C	F767EA54	E62CB2C0
120	1E7024BC	902B9F68	A24C92EE	6CBEE2B7	C2A1C585	6C717B7D	C8E94CC0	662599BE
128	FE811594	A77B9E29	34FE4616	1B62F154	A257D447	6BC40188	FAF258D3	8134AE85
136	1C1DBC7A	EEC79D06	3E934AE5	EA1FDC40	03D7A55C	EFCC114F	B4D80937	8B6B9BC4
144	0EF030D3	A562FA77	03F98EE4	70AA4B5F	2425853A	5B48D70B	7C0C7190	E8DC8213
152	BB12DECB	93D2118C	D2E6D9B1	489B678D	99E9128E	107D0AED	2035CED3	DD374775
160	7402CA34	3964FE67	3DD8B2B0	A9000998	4EE31483	C7650CD3	D280B74F	042B5D15
168	3D23820E	E98C0565	B76F588E	A948ABA8	6548CA95	EB716C83	C749C16A	9FBABD57
176	0CCF8279	03B0B757	AE05FDEA	E1BAC487	F4A4F8C4	0E3E6920	AC92E5AD	A5AB7AD6
184	44F49A8E	3AE05E25	021C229B	C632A8C9	D3EBB6CF	666D051E	B82E9976	0D2208B5
192	4EE29411	3B78DCFC	FBEA1A79	7AEBBF57	288E119D	7F4B95A5	DFA2BFFA	4AC9C904
200	065B5EF2	12661F1C	CD84E7D9	81C6BA35	1A035F30	34D3D379	8D80086E	99E1E3A5
208	1EA99D65	CE34508E	2FC347DF	FBB99BB7	C0772BBD	83BE614E	35730608	BB02E7B3
216	8AC0CA00	152319FA	F9A5B92C	06458374	5055D781	531E0316	032C7EC5	87E2C2B5
224	97383F2C	FF0E6AB7	2FFAEAD9	FF08E36C	E5BC941A	20AA3C5D	B44D06D1	A3AB7974
232	F04571F8	42103A4E	125DF756	2A93B755	99EB8A04	925ADBE9	A1701447	E9AB19F2
240	BF4DD022	05180FEC	1CA9D7F1	2CBBC895	18790EE2	75476695	08A85DAE	A89AB55B
248	2F603725	B7B30E65	A6BEA893	FAC0EC96	540CF545	BBB638F0	4A61447A	ECAC5453
256	208B3149	59B0C224	D53A064F	AB041629	5E0AACEA	6ACB7827	58E15282	8320483D
264	A9F5B3DC	972A3C01	B959E17F	74991BBF	F320F223	CABB896B	22505C7A	405EAA2E
272	2867EC23	BC41EE93	6830CC4E	DF9B3AB6	D2C28789	57173AEF	48F6F780	80FD5978
280	9321D9B6	DF83C22B	79259225	28026B1F	031BCFAA	F340CF70	CA9C1D02	7F8D8DE5
288	6939475D	09D8F12C	E86F9FAF	F2FBF7FE	ABA18BE4	42A35A37	2D81427D	44BAD947
296	CFC81C14	D6F9BCB6	1A309662	3DA31C24	58D3123B	F525FF37	B76408C6	798E3CF1
304	3803025C	51DD5681	03BC328C	8A34CFF9	243D646D	C1C10491	A06D8DE4	247E85C6
312	7312E908	EB37A7EB	CEE84F1E	27707081	E1F36FDF	3F2D7202	668DA038	B7372108
320	820EE0B6	C730BE32	CF1FEF2E	13089821	948CF9E6	242EFA0F	9AB66375	C6B439F1
328	08F8A226	19E65398	09380E43	A92FE5C2	7CFDE834	6F939C42	71D03137	71769ED7
336	21E551C4	58E558C2	AA93D74E	F6C927CD	4BAEF82F	B01135A6	D11B1333	059BD402
344	2BF7B1CA	41A470A5	382168D3	6CDE88AE	51E25A79	EE366131	6316743A	9CC090C2
352	953B0232	73C76723	E0FEB6DA	945C3ECD	B729D733	7BB8E108	8F688075	D9AB3838

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	2374AF7F	3A9B4BA8	060FFD83	BE5FA407	EB3949D7	98D289AD	E3E718D8	28B0EF30
368	90A35841	53922E04	529D26F3	0A4B0B9A	F2CCBFBE	7663EA5B	E647C0A1	E0175945
376	438DCEF1	911B718B	C952F0A4	635F5C92	0C3DA0D9	D091050C	B20DE8B1	620CD827
384	042073B9	BA959787	83A1EA48	98619E14	67B8D23F	C9B509B9	ED645852	DD8B4B81
392	99E138F8	B93AA1E0	9E6E6397	D9E77638	228C406E	75B6D1FF	3E6C5359	B151D7C8
400	2999F9A1	CDFC4D93	8B7A50CB	28C74BF4	877A6E41	480FB902	57161769	C98EA0E4
408	8E718362	1A612140	E195BB50	E42532D3	443219DB	87871265	7B834855	EC7D5B45
416	BCE93349	F6332D04	EF614404	10B3830A	1E094F2F	F88ECEDF	C37BEB31	03351339
424	717335F0	AF5E6788	007BC1AF	A3D19965	31C35E37	A7582CB1	D2F2573B	491AA6A3
432	A42D12E6	C99E810A	6F207154	3F8ECFD1	AA4942AA	BC3A7F8F	1451CA4D	1217EEBC
440	02706911	F39A64DD	962A140E	86887D5E	83563094	3274DE0F	411E6F4E	B52567EA

All values are given in hexadecimal.

The value of h_1 is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
0	C261BE43 471044A7 07F42572 208AFB6F	F579E61F E368F5FD 665D30E5 6125B4EA
	A101CF9F F8D516F4 E1A84F10 77B36148	E31AFAED 1C804E95 5389D02F 1D8919C8
	Input to left $F^4(\cdot)$:	F579E61F E368F5FD 665D30E5 6125B4EA
	Input to right $F^4(\cdot)$:	E31AFAED 1C804E95 5389D02F 1D8919C8
	Output from left $F^4(\cdot)$:	C7B35DD0 CAE08389 D88C79A0 DFEB0FDC
	Output from right $F^4(\cdot)$:	E63F8788 13615BCC C681A05A 825E4332
1	F579E61F E368F5FD 665D30E5 6125B4EA	473E4817 EBB44D38 2729EF4A F5ED227A
	E31AFAED 1C804E95 5389D02F 1D8919C8	05D2E393 8DF0C72E DF785CD2 FF61F4B3
	Input to left $F^4(\cdot)$:	473E4817 EBB44D38 2729EF4A F5ED227A
	Input to right $F^4(\cdot)$:	05D2E393 8DF0C72E DF785CD2 FF61F4B3
	Output from left $F^4(\cdot)$:	D87B12B6 D04C7A10 4E84153E F076C370
	Output from right $F^4(\cdot)$:	A72682FA 9E2EF701 89512F89 8C48D798
2	473E4817 EBB44D38 2729EF4A F5ED227A	443C7817 82AEB994 DAD8FFA6 91C1CE50
	05D2E393 8DF0C72E DF785CD2 FF61F4B3	2D02F4A9 33248FED 28D925DB 9153779A
	Input to left $F^4(\cdot)$:	443C7817 82AEB994 DAD8FFA6 91C1CE50
	Input to right $F^4(\cdot)$:	2D02F4A9 33248FED 28D925DB 9153779A
	Output from left $F^4(\cdot)$:	7E21C3C7 DFD4EE86 EE2EFEAD 64D77F09
	Output from right $F^4(\cdot)$:	BFBECA5B 4FC88256 8B370C8C 4B635300
3	443C7817 82AEB994 DAD8FFA6 91C1CE50	BA6C29C8 C2384578 544F505E B402A7B3
	2D02F4A9 33248FED 28D925DB 9153779A	391F8BD0 3460A3BE C90711E7 913A5D73
	Input to left $F^4(\cdot)$:	BA6C29C8 C2384578 544F505E B402A7B3
	Input to right $F^4(\cdot)$:	391F8BD0 3460A3BE C90711E7 913A5D73
	Output from left $F^4(\cdot)$:	71B4E9C8 C322BE7E D23A8CC6 24F2DE64
	Output from right $F^4(\cdot)$:	DE4A811F 2BAA5AB9 DD346C70 D9E00305
4	BA6C29C8 C2384578 544F505E B402A7B3	F34875B6 188ED554 F5ED49AB 48B3749F
	391F8BD0 3460A3BE C90711E7 913A5D73	358891DF 418C07EA 08E27360 B5331034
	Input to left $F^4(\cdot)$:	F34875B6 188ED554 F5ED49AB 48B3749F
	Input to right $F^4(\cdot)$:	358891DF 418C07EA 08E27360 B5331034
	Output from left $F^4(\cdot)$:	215EFA0E BCBDC599 E4144752 109CEBDC
	Output from right $F^4(\cdot)$:	C33698B8 C03B330A 6BEB519D 58DA68A6
5	F34875B6 188ED554 F5ED49AB 48B3749F	FA291368 F45B90B4 A2EC407A C9E035D5
	358891DF 418C07EA 08E27360 B5331034	9B32D3C6 7E8580E1 B05B170C A49E4C6F
	Input to left $F^4(\cdot)$:	FA291368 F45B90B4 A2EC407A C9E035D5
	Input to right $F^4(\cdot)$:	9B32D3C6 7E8580E1 B05B170C A49E4C6F
	Output from left $F^4(\cdot)$:	504D9961 DA750FB3 8BC5D0D8 16E3E830
	Output from right $F^4(\cdot)$:	6E9F22FA D2370088 0FF71168 BD7F847D
6	FA291368 F45B90B4 A2EC407A C9E035D5	5B17B325 93BB0762 07156208 084C9449
	9B32D3C6 7E8580E1 B05B170C A49E4C6F	A305ECD7 C2FBDAE7 7E289973 5E509CAF
	Input to left $F^4(\cdot)$:	5B17B325 93BB0762 07156208 084C9449
	Input to right $F^4(\cdot)$:	A305ECD7 C2FBDAE7 7E289973 5E509CAF
	Output from left $F^4(\cdot)$:	1E4ECA66 8F3CCD38 0B2D288D FE337B15
	Output from right $F^4(\cdot)$:	0E8820FB EECF641F 2BA536B6 234263BE
7	5B17B325 93BB0762 07156208 084C9449	95BAF33D 904AE4FE 9BFE21BA 87DC2FD1
	A305ECD7 C2FBDAE7 7E289973 5E509CAF	E467D90E 7B675D8C A9C168F7 37D34EC0
	Input to left $F^4(\cdot)$:	95BAF33D 904AE4FE 9BFE21BA 87DC2FD1
	Input to right $F^4(\cdot)$:	E467D90E 7B675D8C A9C168F7 37D34EC0
	Output from left $F^4(\cdot)$:	E25A3E77 D816DFE4 73DA526E 83EBE535
	Output from right $F^4(\cdot)$:	6979089E F2183C2C 362D544D 03904AA1

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
8	95BAF33D 904AE4FE 9BFE21BA 87DC2FD1	CA7CE449 30E3E6CB 4805CD3E 5DC0D60E
	E467D90E 7B675D8C A9C168F7 37D34EC0	B94D8D52 4BADD886 74CF3066 8BA7717C
	Input to left $F^4(\cdot)$:	CA7CE449 30E3E6CB 4805CD3E 5DC0D60E
	Input to right $F^4(\cdot)$:	B94D8D52 4BADD886 74CF3066 8BA7717C
	Output from left $F^4(\cdot)$:	73878083 157E1CDA AA154A01 A2269703
	Output from right $F^4(\cdot)$:	D811621E 618E62A7 09BAF909 302B86E7
9	CA7CE449 30E3E6CB 4805CD3E 5DC0D60E	3C76BB10 1AE93F2B A07B91FE 07F8C827
	B94D8D52 4BADD886 74CF3066 8BA7717C	E63D73BE 8534F824 31EB6BBB 25FAB8D2
	Input to left $F^4(\cdot)$:	3C76BB10 1AE93F2B A07B91FE 07F8C827
	Input to right $F^4(\cdot)$:	E63D73BE 8534F824 31EB6BBB 25FAB8D2
	Output from left $F^4(\cdot)$:	1D715EE2 A67DBD48 87E639D8 2EA1664C
	Output from right $F^4(\cdot)$:	27EEA8FD 33E99A82 51747386 F38F775A
10	3C76BB10 1AE93F2B A07B91FE 07F8C827	9EA325AF 78444204 25BB43E0 78280626
	E63D73BE 8534F824 31EB6BBB 25FAB8D2	D70DBAAB 969E5B83 CFE3F4E6 7361B042
	Input to left $F^4(\cdot)$:	9EA325AF 78444204 25BB43E0 78280626
	Input to right $F^4(\cdot)$:	D70DBAAB 969E5B83 CFE3F4E6 7361B042
	Output from left $F^4(\cdot)$:	D04A5017 78139412 FB7B5956 DEBC0808
	Output from right $F^4(\cdot)$:	F49B570E 02EFDB55 097F5042 7B392EAA
11	9EA325AF 78444204 25BB43E0 78280626	12A624B0 87DB2371 38943BF9 5EC39678
	D70DBAAB 969E5B83 CFE3F4E6 7361B042	EC3CEB07 62FAAB39 5B00C8A8 D944C02F
	Input to left $F^4(\cdot)$:	12A624B0 87DB2371 38943BF9 5EC39678
	Input to right $F^4(\cdot)$:	EC3CEB07 62FAAB39 5B00C8A8 D944C02F
	Output from left $F^4(\cdot)$:	89CDADEF 0DA1ED85 390315C2 116B4567
	Output from right $F^4(\cdot)$:	FC3EFE6F 5737B872 B40F1425 AC540AF9
12	12A624B0 87DB2371 38943BF9 5EC39678	2B3344C4 C1A9E3F1 7BECE0C3 DF35BABB
	EC3CEB07 62FAAB39 5B00C8A8 D944C02F	176E8840 75E5AF81 1CB85622 69434341
	Input to left $F^4(\cdot)$:	2B3344C4 C1A9E3F1 7BECE0C3 DF35BABB
	Input to right $F^4(\cdot)$:	176E8840 75E5AF81 1CB85622 69434341
	Output from left $F^4(\cdot)$:	65D33E2F F6D061F0 7921AABE 71773FCC
	Output from right $F^4(\cdot)$:	BE924CC8 54C14CE0 3191AA22 A6EA949D
13	2B3344C4 C1A9E3F1 7BECE0C3 DF35BABB	52AEA7CF 363BE7D9 6A91628A 7FAE54B2
	176E8840 75E5AF81 1CB85622 69434341	77751A9F 710B4281 41B59147 2FB4A9B4
	Input to left $F^4(\cdot)$:	52AEA7CF 363BE7D9 6A91628A 7FAE54B2
	Input to right $F^4(\cdot)$:	77751A9F 710B4281 41B59147 2FB4A9B4
	Output from left $F^4(\cdot)$:	EDD72489 F68C40F9 48786257 09075BDB
	Output from right $F^4(\cdot)$:	37E28D14 45E3CABF FAE06781 2E27E7DB
14	52AEA7CF 363BE7D9 6A91628A 7FAE54B2	208C0554 3006653E E65831A3 4764A49A
	77751A9F 710B4281 41B59147 2FB4A9B4	C6E4604D 3725A308 33948294 D632E160

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in A Davies-Meyer mode), to produce the output of the compression function:

$$
\begin{aligned}
h_1 = {} & \texttt{C261BE43 471044A7 7F42572 208AFB6F F579E61F E368F5FD 665D30E5 6125B4EA} \\
& \texttt{A101CF9F F8D516F4 E1A84F10 77B36148 E31AFAED 1C804E95 5389D02F 1D8919C8} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \oplus \\
& \texttt{52AEA7CF 363BE7D9 6A91628A 7FAE54B2 208C0554 3006653E E65831A3 4764A49A} \\
& \texttt{77751A9F 710B4281 41B59147 2FB4A9B4 C6E4604D 3725A308 33948294 D632E160} \\
& \qquad\qquad\qquad\qquad\qquad\qquad = \\
h_2 = {} & \texttt{90CF198C 712BA37E 6D6547F8 5F24AFDD D5F5E34B D36E90C3 80050146 26411070} \\
& \texttt{D674D500 89DE5475 A01DDE57 5807C8FC 25FE9AA0 2BA5ED9D 601D52BB CBBBF8A8}
\end{aligned}
$$

This value is truncated to the digest:

$$
\begin{aligned}
&90CF198C\ 712BA37E\ 6D6547F8\ 5F24AFDD\ \ D5F5E34B\ D36E90C3\ 80050146\ 26411070 \\
&D674D500\ 89DE5475\ A01DDE57\ 5807C8FC_x
\end{aligned}
$$

B.4 Digests of 512 Bits

B.4.1 One-Block Message We outline here the internal state of the computation of computing the 512-bit digest of the message M = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

The first step is the padding of the message, yielding the array of 32-bit words:

$$
\begin{aligned}
msg[0,\ldots,31] = {} & \texttt{45434241 48474644 4C4B4A49 504F4E4D 54535251 58575 655} \\
& \texttt{00805A59 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00D00000 00000000} \\
& \texttt{00000000 02000000}_x
\end{aligned}
$$

The message expansion takes this message and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	00805A59	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	00D00000	00000000	00000000	00000000	02000000
32	C6DD21A7	F255D8E5	E9B2B0C3	10B74719	EC42280E	6D0CC7C0	791301B7	609BFE5A
40	8F214B6D	0E6FA4A3	1A7062D4	03F89D39	EC42280E	6D0CC7C0	791301B7	609BFE5A
48	8F214B6D	0E6FA4A3	1A7062D4	03F89D39	EC42280E	6D0CC7C0	791301B7	609BFE5A
56	8F214B6D	3B49B7B0	1A7062D4	03F89D39	EC42280E	582AD4D3	5107158B	609BFE5A
64	FD949617	E825BA31	EA4A2DFA	FCF56F17	B468FCDD	3C0BD24B	1988FFED	9D0F684D
72	6704F15C	E4258959	E6850DC3	B79061E4	D049FA45	7484382D	E41C69FA	079F0F06
80	6B04C234	E8EAA960	ADE00330	D3B1677C	98C61023	8910AE3A	7E8C0EB1	0B9F3C6E
88	67CBE20D	96A9B480	C9C105A8	9B3E8D1A	65528634	26A6DA62	5A9829E5	07501C57
96	716D8B07	559AD213	9E463FA2	C6F9D7B7	6F1B7639	7FDAC5CF	0D03AA1A	4FFB63DB
104	BA27936E	B2A7340A	C089D6F3	A10BD775	2262C3F5	E3CCAFEC	361E51C1	36BAA09D
112	AAAC8017	646DD31E	20ED7442	314D9EC0	9EC3E426	A5F6EF1A	9D4A7BE4	CCFDE54E
120	A793D41D	9C2AE7CD	DB2E2CD5	7ADA983D	1C4F6EF5	E054674C	C4CB377F	72704DDD
128	ED476CCA	8EB4FEC6	E49CA79F	DAB6B942	8F4F1175	BB11F2B0	7F73E7C7	A2BC0F11
136	34936DA8	563B9395	1A3F6FB1	2E44C600	99733145	9CBF482B	94A25ED0	0229CD35
144	FC971382	7E52BCAF	0EA9B242	A83EAF85	027CAC0D	3154B1CA	9F63B6D1	306AF6CC
152	D9C168B2	9283558F	73108350	78A63430	2D1BDF3F	7F37D19D	F4A1C1B3	ABB1256F
160	442B2874	9B86914B	C1A02DFC	FD2D3119	EAC31D89	EEE40B14	6DCFCB28	90646E55
168	7A9DF71F	8A1EB49A	64EAC6F0	56237AF1	4085C03D	0CCEAE67	221E0A54	47A38DBF
176	B95327BF	88E4C6BC	7754FEBC	9CDDF176	AF68FDE1	C6E7BAB1	B90D2314	1773E717
184	5BB7118C	999C0EBC	FD0DF1E9	BAAEF140	F8351006	7F4FBC8C	8005F704	7479A09A
192	DDB726C8	668B60A2	7B0EDCBC	0518211F	958CA105	6EE1FC10	19B66BB2	4DD3489D
200	1C1697BD	F1106826	61F2E7EF	C3AFDBF4	2E643C2D	1578C5D5	6FCD42C9	5BB51A02
208	48434F99	E9162153	B4FB2548	B2B9CD5B	BA103834	A92AF878	E2B83916	5F30A88E
216	B2A130DF	2D672BF4	4FB43CB2	00BEC974	511FE87E	9DF7859A	DF355F8A	C6D89045
224	C495B2B6	FEAFB2D7	5C198984	5E32203D	0A0B7AC6	1F2B216B	AACE2C00	382CEE78
232	1009B5FA	2E42BE45	A57EB5D5	752751FC	04147DA5	E97E0B9E	69EDE4DA	93761DD1
240	7A2624F2	7B25EF01	86A0DEC0	F8D23D2D	A9682C02	E700A2D2	4064908F	34D823EF
248	CD536106	A6705A8A	165C6187	B74AF1E7	436967EF	01225BA2	D7AFA2EB	176F7FF4
256	62E5E83C	E8F3D350	EB537863	1D5B47D2	0B292164	C8848380	BDA153F4	5AC90644
264	F8FA66AA	C511C626	B825F207	7E0E7098	CC90FE25	54DF586A	3324E29E	6B8C7B7B
272	BF37E2D4	C3001D06	F8AEAE58	3442C308	FDB77468	D424404C	2BE8EBF4	8BEFC13B
280	0E537C00	5EDEF4D2	221EA28F	4AFD858F	974D27A3	2ACAB056	5C4063D0	193C03F4
288	DD072C95	60B4B872	E21CC475	48E603D1	6C79ED6D	D8BEB25D	2263C680	09449B28
296	87FE39B1	223D1184	D9C83F86	D9DC9B6C	83EE2CB8	AAE2DD01	E6C2E91B	4F2206F5
304	8E296290	AB272AA8	AD6886D9	2EEBEE0A	95818E21	93F740BF	81B8DA5B	D71178F6
312	47FC34EA	F7496324	D606118E	F4BD2FD1	FC8C30B8	383089A4	CC4D63DD	2774E4B8
320	2A4E4FB1	B6B2A9FC	16A1EBA4	B46A3369	544964C9	14F3D180	05172238	230AD499
328	314C904D	349CFA20	6DA20CEF	8D95FFA5	971DFD38	AFF5FF39	C5C83D82	7E6E96B8
336	BAB598B0	C6852647	20FD797C	B9F61332	3A747118	563F7D3D	FFD64CE3	6DA4E046
344	817912AD	D7B41A58	6FF002BC	CEC95EC9	AAB34D85	C7E6C547	A1E9839B	A60DF615
352	6B0FEF2D	7B825873	4B1D4779	3E5FFD16	1A0B0921	8A2F9E6B	73082F18	1425C42F

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	91BE39F8	5B508A10	40AF7B5E	005A3051	D7A09A05	F03D2D0E	4F2635C4	7100A4CB
368	C2DCB23E	A564AF93	60D3733C	B523D5AB	F60FC0A0	D639C40D	5BDB0B37	42689CD8
376	A87F5657	421A862E	50BFC66F	7C218F97	C8D4A86F	07034517	1DA4E4E0	03FC85A8
384	29156903	2B3D9E1C	373CC8EE	F68B5579	1D084C36	978B7A8B	70F4AAB0	3D30AD2C
392	BA83A7E4	6C6C42FE	B6242E27	1D527C67	402BE08E	80C987BE	721698E8	CB83032F
400	AEB0F0C0	134081B4	7D810F5B	F5083525	76C6471E	A42F5CE5	90580818	ECD86C18
408	BB3FD7E3	3F9B8975	A5B7F34A	0AE7C889	6CFBF48A	975B4D0F	F17C88F8	B8C3524B
416	34A38063	9A37BC5D	BDB9FD3C	DEBDDA07	75EF89C0	2028FCF9	9E6C776A	17FD9383
424	F68A74C4	5B39DC1A	02B328DE	A07FC1F4	DC5D2386	BB2C4090	B8F75E34	9D08A65E
432	1E23B0F4	0FB10037	68E90C9C	1C5BE668	103CA5F1	15312AFF	CF270E6A	CEB44744
440	E332810D	2D8B189E	EF8E28A6	97C6512D	A6E49B62	CFDA681B	8483557C	0B00655D

All values are given in hexadecimal.

The value of IV_{512} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i) Middle Rifht (B_i)	Middle Left (A_i) Right half (R_i)
0	8A671C48 21FBB075 6C11F5A0 2B153831	C6192444 1254BA09 ADBD2BF9 6956353E
	51ECE04E B38D02EC 3CCCC57B B76EA6DA	DDED39A5 ACB431B4 9452E478 F2DCEE8D
	Input to left $F^4(\cdot)$:	C6192444 1254BA09 ADBD2BF9 6956353E
	Input to right $F^4(\cdot)$:	DDED39A5 ACB431B4 9452E478 F2DCEE8D
	Output from left $F^4(\cdot)$:	704B874A 56FF711F 397CDF4B 8DA9DF34
	Output from right $F^4(\cdot)$:	82B8D9BC 8FA3A907 CBC5DE1A 1A072889
1	C6192444 1254BA09 ADBD2BF9 6956353E	D35439F2 3C2EABEB F7091B61 AD698E53
	DDED39A5 ACB431B4 9452E478 F2DCEE8D	FA2C9B02 7704C16A 556D2AEB A6BCE705
	Input to left $F^4(\cdot)$:	D35439F2 3C2EABEB F7091B61 AD698E53
	Input to right $F^4(\cdot)$:	FA2C9B02 7704C16A 556D2AEB A6BCE705
	Output from left $F^4(\cdot)$:	44F45757 E9CD6B81 D25C0504 6EEDD899
	Output from right $F^4(\cdot)$:	6E20639F 4FED90A4 67E9F1DE FDADDDAF
2	D35439F2 3C2EABEB F7091B61 AD698E53	B3CD5A3A E359A110 F3BB15A6 0F713322
	FA2C9B02 7704C16A 556D2AEB A6BCE705	82ED7313 FB99D188 7FE12EFD 07BBEDA7
	Input to left $F^4(\cdot)$:	B3CD5A3A E359A110 F3BB15A6 0F713322
	Input to right $F^4(\cdot)$:	82ED7313 FB99D188 7FE12EFD 07BBEDA7
	Output from left $F^4(\cdot)$:	D5026101 4AF77A76 3D414DBA FD051002
	Output from right $F^4(\cdot)$:	2F975707 49BB1C22 BCA2C544 9BE29429
3	B3CD5A3A E359A110 F3BB15A6 0F713322	D5BBCC05 3EBFDD48 E9CFEFAF 3D5E732C
	82ED7313 FB99D188 7FE12EFD 07BBEDA7	065658F3 76D9D19D CA4856DB 506C9E51
	Input to left $F^4(\cdot)$:	D5BBCC05 3EBFDD48 E9CFEFAF 3D5E732C
	Input to right $F^4(\cdot)$:	065658F3 76D9D19D CA4856DB 506C9E51
	Output from left $F^4(\cdot)$:	477EEC01 BCD7A82B 8D1A98E2 9CC0E968
	Output from right $F^4(\cdot)$:	5B87A306 6438A4BB 9161B089 3211B4DD
4	D5BBCC05 3EBFDD48 E9CFEFAF 3D5E732C	D96AD015 9FA17533 EE809E74 35AA597A
	065658F3 76D9D19D CA4856DB 506C9E51	F4B3B63B 5F8E093B 7EA18D44 93B1DA4A
	Input to left $F^4(\cdot)$:	D96AD015 9FA17533 EE809E74 35AA597A
	Input to right $F^4(\cdot)$:	F4B3B63B 5F8E093B 7EA18D44 93B1DA4A
	Output from left $F^4(\cdot)$:	FD613785 617773DB 796112CD 89AA3E8F
	Output from right $F^4(\cdot)$:	213BCDFC B565FDF7 A4C52021 908745B6
5	D96AD015 9FA17533 EE809E74 35AA597A	276D950F C3BC2C6A 6E8D76FA C0EBDBE7
	F4B3B63B 5F8E093B 7EA18D44 93B1DA4A	28DAFB80 5FC8AE93 90AEFD62 B4F44DA3
	Input to left $F^4(\cdot)$:	276D950F C3BC2C6A 6E8D76FA C0EBDBE7
	Input to right $F^4(\cdot)$:	28DAFB80 5FC8AE93 90AEFD62 B4F44DA3
	Output from left $F^4(\cdot)$:	1BB9DB24 1CACA534 35D43C62 5D9E9713
	Output from right $F^4(\cdot)$:	20B606D2 0770565A A7B39687 AE164E6F
6	276D950F C3BC2C6A 6E8D76FA C0EBDBE7	D405B0E9 58FE5F61 D9121BC3 3DA79425
	28DAFB80 5FC8AE93 90AEFD62 B4F44DA3	C2D30B31 830DD007 DB54A216 6834CE69
	Input to left $F^4(\cdot)$:	D405B0E9 58FE5F61 D9121BC3 3DA79425
	Input to right $F^4(\cdot)$:	C2D30B31 830DD007 DB54A216 6834CE69
	Output from left $F^4(\cdot)$:	334E8302 36264D30 6A002798 1D577F37
	Output from right $F^4(\cdot)$:	A3EFA0C9 3A3A0CDB C31B1CFA 32A93B35
7	D405B0E9 58FE5F61 D9121BC3 3DA79425	8B355B49 65F2A248 53B5E198 865D7696
	C2D30B31 830DD007 DB54A216 6834CE69	1423160D F59A615A 048D5162 DDBCA4D0
	Input to left $F^4(\cdot)$:	8B355B49 65F2A248 53B5E198 865D7696
	Input to right $F^4(\cdot)$:	1423160D F59A615A 048D5162 DDBCA4D0
	Output from left $F^4(\cdot)$:	C73F263B 4C02F9DE A425653F CAB63C20
	Output from right $F^4(\cdot)$:	124B9B69 B02E519F 1E4373F4 B6ADA930

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i) Middle Rifht (B_i)	Middle Left (A_i) Right half (R_i)
8	8B355B49 65F2A248 53B5E198 865D7696	D0989058 33238198 C517D1E2 DE996759
	1423160D F59A615A 048D5162 DDBCA4D0	133A96D2 14FCA6BF 7D377EFC F711A805
	Input to left $F^4(\cdot)$:	D0989058 33238198 C517D1E2 DE996759
	Input to right $F^4(\cdot)$:	133A96D2 14FCA6BF 7D377EFC F711A805
	Output from left $F^4(\cdot)$:	5C38E9EC 98D8B11D 15E0562C EA4C850D
	Output from right $F^4(\cdot)$:	8FD6158C 27D04FE6 CB7AAB82 731B2D0E
9	D0989058 33238198 C517D1E2 DE996759	9BF50381 D24A2EBC CFF7FAE0 AEA789DE
	133A96D2 14FCA6BF 7D377EFC F711A805	D70DB2A5 FD2A1355 4655B7B4 6C11F39B
	Input to left $F^4(\cdot)$:	9BF50381 D24A2EBC CFF7FAE0 AEA789DE
	Input to right $F^4(\cdot)$:	D70DB2A5 FD2A1355 4655B7B4 6C11F39B
	Output from left $F^4(\cdot)$:	9C006585 46DA1081 05C600B3 45DBA130
	Output from right $F^4(\cdot)$:	3982691F 82874380 7200A508 FA349A3F
10	9BF50381 D24A2EBC CFF7FAE0 AEA789DE	2AB8FFCD 967BE53F 0F37DBF4 0D25323A
	D70DB2A5 FD2A1355 4655B7B4 6C11F39B	4C98F5DD 75F99119 C0D1D151 9B42C669
	Input to left $F^4(\cdot)$:	2AB8FFCD 967BE53F 0F37DBF4 0D25323A
	Input to right $F^4(\cdot)$:	4C98F5DD 75F99119 C0D1D151 9B42C669
	Output from left $F^4(\cdot)$:	AA033F64 8B083765 D037B010 2D1A95D3
	Output from right $F^4(\cdot)$:	19F6518E 4700B67E B19B86D9 85B66163
11	2AB8FFCD 967BE53F 0F37DBF4 0D25323A	CEFBE32B BA2AA52B F7CE316D E9A792F8
	4C98F5DD 75F99119 C0D1D151 9B42C669	31F63CE5 594219D9 1FC04AF0 83BD1C0D
	Input to left $F^4(\cdot)$:	CEFBE32B BA2AA52B F7CE316D E9A792F8
	Input to right $F^4(\cdot)$:	31F63CE5 594219D9 1FC04AF0 83BD1C0D
	Output from left $F^4(\cdot)$:	C94D4096 3D02D462 FE9E2D62 7AF6E6E7
	Output from right $F^4(\cdot)$:	116EB172 FC9094BF C357DE5D 4BEA53A6
12	CEFBE32B BA2AA52B F7CE316D E9A792F8	5DF644AF 896905A6 03860F0C D0A895CF
	31F63CE5 594219D9 1FC04AF0 83BD1C0D	E3F5BF5B AB79315D F1A9F696 77D3D4DD
	Input to left $F^4(\cdot)$:	5DF644AF 896905A6 03860F0C D0A895CF
	Input to right $F^4(\cdot)$:	E3F5BF5B AB79315D F1A9F696 77D3D4DD
	Output from left $F^4(\cdot)$:	FEA1605C 6FE16975 0E28645D BC344E8A
	Output from right $F^4(\cdot)$:	6C2A5AD0 C844F8C3 035304D1 E704074A
13	5DF644AF 896905A6 03860F0C D0A895CF	5DDC6635 9106E11A 1C934E21 64B91B47
	E3F5BF5B AB79315D F1A9F696 77D3D4DD	305A8377 D5CBCC5E F9E65530 5593DC72
	Input to left $F^4(\cdot)$:	5DDC6635 9106E11A 1C934E21 64B91B47
	Input to right $F^4(\cdot)$:	305A8377 D5CBCC5E F9E65530 5593DC72
	Output from left $F^4(\cdot)$:	4DCBCE62 825B7828 8030F1E6 5EB2E2EB
	Output from right $F^4(\cdot)$:	56D6DD34 F12D6E07 BD93FF2E BE342DC3
14	5DDC6635 9106E11A 1C934E21 64B91B47	B523626F 5A545F5A 4C3A09B8 C9E7F91E
	305A8377 D5CBCC5E F9E65530 5593DC72	103D8ACD 0B327D8E 83B6FEEA 8E1A7724

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$h_0 =$ 8A671C48 21FBB075 6C11F5A0 2B153831 C6192444 1254BA09 ADBD2BF9 6956353E
51ECE04E B38D02EC 3CCCC57B B76EA6DA DDED39A5 ACB431B4 9452E478 F2DCEE8D
$\oplus$
5DDC6635 9106E11A 1C934E21 64B91B47 B523626F 5A545F5A 4C3A09B8 C9E7F91E
305A8377 D5CBCC5E F9E65530 5593DC72 103D8ACD 0B327D8E 83B6FEEA 8E1A7724
$=$
$h_1 =$ D7BB7A7D B0FD516F 7082BB81 4FAC2376 733A462B 4800E553 E1872241 A0B1CC20
61B66339 6646CEB2 C52A904B E2FD7AA8 CDD0B368 A7864C3A 17E41A92 7CC699A9

This value is the digest:

$D7BB7A7D$ $B0FD516F$ $7082BB81$ $4FAC2376$ $733A462B$ $4800E553$ $E1872241$ $A0B1CC20$
$61B66339$ $6646CEB2$ $C52A904B$ $E2FD7AA8$ $CDD0B368$ $A7864C3A$ $17E41A92$ $7CC699A9$

B.4.2 Two-Block Message For a two block message, we picked the message

$M =$ "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq
rstuvwxyz01234567890123456789ABCDEFGHIJKLMNOPQR
STUVWXYZabcdefghijklmnopqrstuvwxyz0123456789012
3456789ABCDEFGHIJKLMNOP".

The first message block, is therefore,

$msg[0,\ldots,31] =$ 45434241 48474644 4C4B4A49 504F4E4D 54535251 58575 655
62615A59 66646563 6A696867 6E6D6C6B 7271706F 76757473
7A797877 33323130 37363534 31303938 35343332 39383736
45434241 48474644 4C4B4A49 504F4E4D 54535251 58575655
62615A59 66656463 6A696867 6E6D6C6B 7271706F 76757473
$7A797877\ 33323130_x$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655	62615A59	66656463
8	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130	37363534	31303938
16	35343332	39383736	45434241	48474644	4C4B4A49	504F4E4D	54535251	58575655
24	62615A59	66656463	6A696867	6E6D6C6B	7271706F	76757473	7A797877	33323130
32	B4AC5518	8420AC96	93CBC8B4	21857629	D8173EB3	E0EC54DD	A33A2930	C9317C41
40	14FDA929	E63D823B	3BB3AB9A	57376A2F	61F74F91	7D74733E	B0496789	0C0DADE1
48	121C96A1	9C1A2FE5	251D9CE0	05FA495D	127D1C7E	040EDDD5	C1AA737B	3C3CFB41
56	FF9C300F	015692BC	08B0117C	9EBC1752	2129C87D	15B643BE	0C764AE2	1A06A5BB
64	B5FAC7A4	8C90BDEA	0D77DFE6	00ACBE54	CDA17D0D	EC9A1E3F	B93C8C8B	7CCBBBE5
72	986D14C3	EB4A5DDD	3B1F15CE	9A961722	8D6D51AE	C448FFB5	CC82DC6C	9460B922
80	F956CB7C	A7053A2B	BF8B8BC2	889718F3	D635E3CB	C88C01B9	55CACA59	C56A303D
88	58990A24	BEDD197E	8027098F	4889F499	E9A5C9C4	407C89E7	C91C7ADF	429FAF9F
96	C290C4A1	1C44CAC7	8C25492A	ED135D45	5E614C5C	1B43F937	B5FAC820	E2B0E0EC
104	1236EC10	544E3CF9	5D65528E	C5396C94	AD1278A2	AD037A94	1B559062	9CC2DA9A
112	EFC868F6	D0D0CCE2	E556914E	38C74566	A3DBDD9B	724DEA42	A4881FD8	30425639
120	221A806D	958F4033	8E0985F2	535CD680	EAA7C3AB	0B17407A	E5454300	5ECC01C5
128	571F8492	924D4F35	DF799FAA	07B49EEE	55760C26	FE06BA37	EB36C9E5	B5AF647E
136	807BA325	8B37A353	5AD1CC60	904F60B2	5314C295	4635B371	AEFAF41C	1CB979BF
144	64FFCBA5	8A010082	7519F1FC	6BD387F3	E5EE6EEA	DCB71E5E	B8316667	54BD9D9C
152	A81B80EF	E096B1CF	E5DA0201	B6B2B86A	3610DDF5	B326261D	B1F8DE9C	F6D7812A
160	2D7AE307	D1CE2A41	539D27CF	8D0FECAF	FBF9D962	F756E8C9	87E649CE	F1A2D4FF
168	2CEC20B5	E008EA5F	FE0A77FE	B9AA4C0C	73075E94	4F8CEC46	395274F5	6EA24A19
176	F9F5D0D9	C8EF1CA5	2E14107A	C0EC3B1B	6583BFC4	5995C744	E1776851	64F75456
184	F6A7AB62	EEE42DC1	2343861C	2595EEE8	05BAAE4F	12CC851E	CB528240	D5A712DC
192	C39ECEC6	F28DAC5D	7608C927	88B542E0	E9355C7C	3C046A89	52415B12	323C1A39
200	DE618CE8	96002378	76BF351E	509F1070	4F03341D	1DCDB754	0B6E6ECC	B0C3C6F1
208	6FF5F3A1	BE5029BB	7E8B000A	8FEF0F06	784E0890	52FBA988	51B4AEA0	0002A7F7
216	48F782D9	906F2DCB	ACAC891A	5DDBE678	574107C7	43782BBE	C05025B7	9D509005
224	96682128	903B1F30	13E69CF9	1A597A2E	7773260F	5B8F92C4	3F3D6D81	00E1D2E3
232	052D874B	850A6650	E7B30B59	58CA6893	FCCF38C4	9975335B	5D09032F	A9348C7F
240	B8B31102	F9DE2048	483E6E37	1BF00072	08368413	2D75B1F4	5CC5EC6D	E50DD896
248	DBD32DC6	C8A79EC8	06889E56	B95AA80C	BC09F54C	39410FCC	7C6A5A8F	84488CB4
256	5ECFBFE0	96B38166	AABC34F5	A6508F62	4E3229C3	27E5C84B	BB75E135	5E2E6D03
264	939E062D	2FB652A5	41E3843B	16F84150	DB2AF08F	2200D26E	03276E2C	3AAA8A52
272	970543A7	B83DA473	5EC62F67	C0DAF0FD	2A36567D	2E52DFD8	666F663F	72089B31
280	63EE89B5	9661B1AF	C6526EAB	936CFE71	925B2A94	5F2E69F3	0E62C1BE	E7A60501
288	947501F0	D20FD321	29BB4683	52A6E5D3	B9F25A42	58C097ED	82105797	FF487575
296	FB833044	EF768151	F3A3268A	82BAACBD	846206BE	E55EEFD9	1847A1A3	D17F5391
304	4BAE398A	39095E79	06757645	F2D22A88	21179280	DFF3876F	62F38BF4	D7A5FB1A
312	74840B21	826CBDAD	135E52AF	2C823421	5D9BDEEE	4BB6F490	3C06FB8F	E55B3E41
320	1619BC5D	C151818E	053972A2	0F3D3B3D	F244AED2	64C66C62	674B69D6	E951C928
328	3AD2B1CA	EA4FF3F3	FC9E1DB7	70FE026F	E0A46ADC	8215860F	F116688B	EBADE25B
336	A1E1CA79	C59743CE	768B742A	12764054	A302148F	2EE5EFE4	895E69AF	76443163
344	B11348EF	F4E7C987	012812FB	8F8020AE	737E310A	C2E89D3F	4A42CAEC	544876AE
352	1B5350EA	017B30F2	F7A93EDA	D55109B8	3BA4C530	36DA521C	40879A44	D68F811D

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	30D83B36	5E49C5BE	82198446	564335ED	C170D5CB	D7A96F3D	E993EB11	18E64CBB
368	6E019D79	10F60134	8C7ADA62	4D52A9B7	50A1F453	FB1BE9F5	C3AFBB70	2B1A8564
376	016B70AF	0A07C5C0	7055055B	2A4567A9	6039A01A	F9524BDE	AEA42311	099FAC1F
384	1154952A	712E35A9	DDEC5973	B568A9A2	C2F68EEE	987E710D	4918365B	C7DB1437
392	41F60E9F	83A59CCD	37712DE4	94B5BB03	590EA4C6	9EB15966	2E48FF26	59104224
400	EDA401B4	27872CD0	18CF6161	145C0D71	CE10AD35	D55316D3	9ABFF954	C6BE84D0
408	26EC5C7F	12C8A4A1	6409082A	E455CA9C	B56AB6C9	63EDB28A	681AA7C1	2F73F060
416	97C552FB	CC0B0C00	A0A1EF16	92EF8073	25FF2E36	A4A5DE58	6B247DD4	49A2E9C5
424	29B0D8FD	F9D500F2	5E9201FC	FD254A73	51DF1EED	A4833C44	09A5EE2F	13693F87
432	0DF361FE	374EB37E	A3B195C7	CC9B46DD	DA97759B	46733E5A	F26305CD	041199CF
440	F6123811	A73F474D	C5B30420	3CA39FB2	B7E41E88	97D393BF	A683F5DA	FEE8CC64

All values are given in hexadecimal.

The value of IV_{512} is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
0	8A671C48 21FBB075 6C11F5A0 2B153831	C6192444 1254BA09 ADBD2BF9 6956353E
	51ECE04E B38D02EC 3CCCC57B B76EA6DA	DDED39A5 ACB431B4 9452E478 F2DCEE8D
	Input to left $F^4(\cdot)$:	C6192444 1254BA09 ADBD2BF9 6956353E
	Input to right $F^4(\cdot)$:	DDED39A5 ACB431B4 9452E478 F2DCEE8D
	Output from left $F^4(\cdot)$:	A86801EC CDB3CAB4 5E8A4D6F EC041547
	Output from right $F^4(\cdot)$:	78650CD4 C8EC66F4 9EBCFE09 8AEEAED6
1	C6192444 1254BA09 ADBD2BF9 6956353E	2989EC9A 7B616418 A2703B72 3D80080C
	DDED39A5 ACB431B4 9452E478 F2DCEE8D	220F1DA4 EC487AC1 329BB8CF C7112D76
	Input to left $F^4(\cdot)$:	2989EC9A 7B616418 A2703B72 3D80080C
	Input to right $F^4(\cdot)$:	220F1DA4 EC487AC1 329BB8CF C7112D76
	Output from left $F^4(\cdot)$:	45322677 8CFDB1B7 6E1C1CB7 43B3D961
	Output from right $F^4(\cdot)$:	FF63865E 55F5DB50 55EBB3BE 49EC4C6A
2	2989EC9A 7B616418 A2703B72 3D80080C	228EBFFB F941EAE4 C1B957C6 BB30A2E7
	220F1DA4 EC487AC1 329BB8CF C7112D76	832B0233 9EA90BBE C3A1374E 2AE5EC5F
	Input to left $F^4(\cdot)$:	228EBFFB F941EAE4 C1B957C6 BB30A2E7
	Input to right $F^4(\cdot)$:	832B0233 9EA90BBE C3A1374E 2AE5EC5F
	Output from left $F^4(\cdot)$:	3FDC3C47 C4B48D41 0C1B67BA 446614D9
	Output from right $F^4(\cdot)$:	6E9C6D79 5D9123C5 C9641A71 E6BC025F
3	228EBFFB F941EAE4 C1B957C6 BB30A2E7	4C9370DD B1D95904 FBFFA2BE 21AD2F29
	832B0233 9EA90BBE C3A1374E 2AE5EC5F	1655D0DD BFD5E959 AE6B5CC8 79E61CD5
	Input to left $F^4(\cdot)$:	4C9370DD B1D95904 FBFFA2BE 21AD2F29
	Input to right $F^4(\cdot)$:	1655D0DD BFD5E959 AE6B5CC8 79E61CD5
	Output from left $F^4(\cdot)$:	B143FE5D 6A7DB308 F3DC6035 CCFEA421
	Output from right $F^4(\cdot)$:	493BBE1F 22318862 0C971920 0FD1DE22
4	4C9370DD B1D95904 FBFFA2BE 21AD2F29	CA10BC2C BC9883DC CF362E6E 2534327D
	1655D0DD BFD5E959 AE6B5CC8 79E61CD5	93CD41A6 933C59EC 326537F3 77CE06C6
	Input to left $F^4(\cdot)$:	CA10BC2C BC9883DC CF362E6E 2534327D
	Input to right $F^4(\cdot)$:	93CD41A6 933C59EC 326537F3 77CE06C6
	Output from left $F^4(\cdot)$:	C27EF077 935ADA3F 3D589A85 DE162F50
	Output from right $F^4(\cdot)$:	189BB917 E0CBD8D8 C296E3D2 77A717F3
5	CA10BC2C BC9883DC CF362E6E 2534327D	0ECE69CA 5F1E3181 6CFDBF1A 0E410B26
	93CD41A6 933C59EC 326537F3 77CE06C6	8EED80AA 2283833B C6A7383B FFBB0079
	Input to left $F^4(\cdot)$:	0ECE69CA 5F1E3181 6CFDBF1A 0E410B26
	Input to right $F^4(\cdot)$:	8EED80AA 2283833B C6A7383B FFBB0079
	Output from left $F^4(\cdot)$:	DB5689B8 9C2EA294 226BF32E CDDB3D14
	Output from right $F^4(\cdot)$:	F68E032E B85968A1 43044EDD BF70BD65
6	0ECE69CA 5F1E3181 6CFDBF1A 0E410B26	65434288 2B65314D 7161792E C8BEBBA3
	8EED80AA 2283833B C6A7383B FFBB0079	11463594 20B62148 ED5DDD40 E8EF0F69
	Input to left $F^4(\cdot)$:	65434288 2B65314D 7161792E C8BEBBA3
	Input to right $F^4(\cdot)$:	11463594 20B62148 ED5DDD40 E8EF0F69
	Output from left $F^4(\cdot)$:	9D4515C3 201BD045 BB359713 56E4EA24
	Output from right $F^4(\cdot)$:	9CDAB7EB 2C7E7015 270B960A 610B26FF
7	65434288 2B65314D 7161792E C8BEBBA3	12373741 0EFDF32E E1ACAE31 9EB02686
	11463594 20B62148 ED5DDD40 E8EF0F69	938B7C09 7F05E1C4 D7C82809 58A5E102
	Input to left $F^4(\cdot)$:	12373741 0EFDF32E E1ACAE31 9EB02686
	Input to right $F^4(\cdot)$:	938B7C09 7F05E1C4 D7C82809 58A5E102
	Output from left $F^4(\cdot)$:	B8D95730 4ACD1AC3 1B814F3D CFAB9FD0
	Output from right $F^4(\cdot)$:	C7641D3A 6EC72D47 924FEF76 BA542F33

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i)	Middle Left (A_i)
	Middle Rifht (B_i)	Right half (R_i)
8	12373741 0EFDF32E E1ACAE31 9EB02686	D62228AE 4E710C0F 7F123236 52BB205A
	938B7C09 7F05E1C4 D7C82809 58A5E102	DD9A15B8 61A82B8E 6AE03613 07152473
	Input to left $F^4(\cdot)$:	D62228AE 4E710C0F 7F123236 52BB205A
	Input to right $F^4(\cdot)$:	DD9A15B8 61A82B8E 6AE03613 07152473
	Output from left $F^4(\cdot)$:	8E486B62 58300FAE D6E0D13A D470573D
	Output from right $F^4(\cdot)$:	DA3E69BF BA3D2431 63D19C43 300BB5BA
9	D62228AE 4E710C0F 7F123236 52BB205A	49B515B6 C538C5F5 B419B44A 68AE54B8
	DD9A15B8 61A82B8E 6AE03613 07152473	9C7F5C23 56CDFC80 374C7F0B 4AC071BB
	Input to left $F^4(\cdot)$:	49B515B6 C538C5F5 B419B44A 68AE54B8
	Input to right $F^4(\cdot)$:	9C7F5C23 56CDFC80 374C7F0B 4AC071BB
	Output from left $F^4(\cdot)$:	88552FE0 E11B859A 03858EDC 0D6FDB10
	Output from right $F^4(\cdot)$:	D7A65BBC AB843365 421D4E16 C6B4ECD8
10	49B515B6 C538C5F5 B419B44A 68AE54B8	0A3C4E04 CA2C18EB 28FD7805 C1A1C8AB
	9C7F5C23 56CDFC80 374C7F0B 4AC071BB	5E77074E AF6A8995 7C97BCEA 5FD4FB4A
	Input to left $F^4(\cdot)$:	0A3C4E04 CA2C18EB 28FD7805 C1A1C8AB
	Input to right $F^4(\cdot)$:	5E77074E AF6A8995 7C97BCEA 5FD4FB4A
	Output from left $F^4(\cdot)$:	11189431 EC3E16F4 91424038 3C9C02F9
	Output from right $F^4(\cdot)$:	21EFDDBC 3DE5E754 C1B5BEAA 52A7B3C2
11	0A3C4E04 CA2C18EB 28FD7805 C1A1C8AB	BD90819F 6B281BD4 F6F9C1A1 1867C279
	5E77074E AF6A8995 7C97BCEA 5FD4FB4A	58AD8187 2906D301 255BF472 54325641
	Input to left $F^4(\cdot)$:	BD90819F 6B281BD4 F6F9C1A1 1867C279
	Input to right $F^4(\cdot)$:	58AD8187 2906D301 255BF472 54325641
	Output from left $F^4(\cdot)$:	3BC4B50F 7B42B971 0ED6142C A0AC58F7
	Output from right $F^4(\cdot)$:	54B3760D F07C6137 0032E7AC 88D4CE80
12	BD90819F 6B281BD4 F6F9C1A1 1867C279	0AC47143 5F16E8A2 7CA55B46 D70035CA
	58AD8187 2906D301 255BF472 54325641	31F8FB0B B16EA19A 262B6C29 610D905C
	Input to left $F^4(\cdot)$:	0AC47143 5F16E8A2 7CA55B46 D70035CA
	Input to right $F^4(\cdot)$:	31F8FB0B B16EA19A 262B6C29 610D905C
	Output from left $F^4(\cdot)$:	5154EC8E 50584E95 1579F7A3 514110AB
	Output from right $F^4(\cdot)$:	2A248CDF A28E1760 3221992D 9140C6F4
13	0AC47143 5F16E8A2 7CA55B46 D70035CA	72890D58 8B88C461 177A6D5F C57290B5
	31F8FB0B B16EA19A 262B6C29 610D905C	ECC46D11 3B705541 E3803602 4926D2D2
	Input to left $F^4(\cdot)$:	72890D58 8B88C461 177A6D5F C57290B5
	Input to right $F^4(\cdot)$:	ECC46D11 3B705541 E3803602 4926D2D2
	Output from left $F^4(\cdot)$:	78584897 5E9664EC 021F7103 DDAF5134
	Output from right $F^4(\cdot)$:	9A5E624C E6B16B7C 8FFB23AD B3ADC5CB
14	72890D58 8B88C461 177A6D5F C57290B5	ABA69947 57DFCAE6 A9D04F84 D2A05597
	ECC46D11 3B705541 E3803602 4926D2D2	729C39D4 01808C4E 7EBA2A45 0AAF64FE

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$$
\begin{aligned}
h_0 = {} & \texttt{8A671C48 21FBB075 6C11F5A0 2B153831 C6192444 1254BA09 ADBD2BF9 6956353E} \\
& \texttt{51ECE04E B38D02EC 3CCCC57B B76EA6DA DDED39A5 ACB431B4 9452E478 F2DCEE8D} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \oplus \\
& \texttt{72890D58 8B88C461 177A6D5F C57290B5 ABA69947 57DFCAE6 A9D04F84 D2A05597} \\
& \texttt{ECC46D11 3B705541 E3803602 4926D2D2 729C39D4 01808C4E 7EBA2A45 0AAF64FE} \\
& \qquad\qquad\qquad\qquad\qquad\qquad = \\
h_1 = {} & \texttt{F8EE1110 AA737414 7B6B98FF EE67A884 6DBFBD03 458B70EF 046D647D BBF660A9} \\
& \texttt{BD288D5F 88FD57AD DF4CF379 FE487408 AF710071 AD34BDFA EAE8CE3D F8738A73}
\end{aligned}
$$

The second message block after padding is:

$$
\begin{aligned}
\text{msg}[0,\ldots,31] = {} & \texttt{37363534 31303938 35343332 39383736 00000080 0000000 0} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 04800000 00000000 00000000} \\
& \texttt{00000000 02000000}_x
\end{aligned}
$$

The message expansion takes this block and transforms it into 144 32-bit words stored in the $rk[\cdot]$ array:

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
0	37363534	31303938	35343332	39383736	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
16	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	04800000	00000000	00000000	00000000	02000000
32	3F5ADC60	AF3EAF3E	1A0D5667	E7C90C78	F2975644	CC5DCC5D	796E3504	84AA6F1B
40	91F43527	AF3EAF3E	1A0D5667	E7C90C78	F2975644	CC5DCC5D	796E3504	84AA6F1B
48	91F43527	AF3EAF3E	1A0D5667	E7C90C78	F2975644	CC5DCC5D	796E3504	84AA6F1B
56	91F43527	46790190	239CC7CF	E7C90C78	F2975644	251A62F3	68EBB090	84AA6F1B
64	7923DDF0	8CA268F1	FDC45A1F	155E5A3C	D78D34B7	A4B67CCD	FDC45A1F	FD89B2EB
72	1D565DD6	52FAF521	0F530C5B	304438CF	56212A89	31999642	84E787EF	99FC32CD
80	C30EC006	A06DA365	2A496EA8	B1E826F1	C30EC006	48BA4BB2	E09207C9	47A4AF1D
88	31999642	6C306F38	9274E13E	24C7CC7E	BA2D1DF6	C588653A	2F4F1F8D	B533F959
96	64248DFB	E5950806	7255B19E	D9C334D1	4CA55642	1A011724	ED4BC3B4	406133FA
104	919BB45F	6A68FE08	F23EEA9D	100CD8D1	27AD53EF	D654B4E4	6645B87C	EAFE7581
112	2956CCC6	4604726B	C31C9201	CCB3BA09	4A7E92D8	C995ECCE	10E29843	F69E0619
120	20E0B2A9	91E15747	4F42C620	64B1204A	3BA77FC1	C11E4904	9DB6B66B	CB46CDC4
128	F5C5DABC	AAD7CE26	16E491D4	E2644B10	8DBB1F46	87B7A14F	260D0E70	B5A4E946
136	3B4C7A79	7C8C6FDC	105AA18D	9DB7C797	A01AF2A0	F059BA94	D3E1513A	D1B20FF8
144	55DAA31A	565ED3E6	5EAB5596	6CA948A9	BA27284C	1A74BDF4	C15097BB	A344A503
152	76BE614F	CF4A02D1	23EB8E89	DE960806	21D3C235	004EDEBF	3EF21368	BDF8AC8B
160	0F06D5CA	EB817697	27F3F866	335018BA	07DCC78E	813BC40F	D0EE5078	98F84B6B
168	8E4C3FB1	B24B992E	E25C8BA1	510C2A0B	08519C6B	8CE1A77E	17BA7FAE	5C487BA8
176	277016CC	D694B148	B87E1F54	78C852AE	A2579036	8039608A	73930833	D5FBB245
184	39C00479	B63051E3	70F7A798	B230DF1C	789476F3	2767F05A	76234EE7	6229B028
192	B9368429	9B76D10F	95C3277A	4BC46E49	20BB37D4	F7188AE8	B2C7E050	21CECF42
200	153AEEBE	2788BE54	A998E5E8	71B71DDF	FF491683	3E26472E	3674B0EC	49729516
208	00F8A898	7F0C54A0	C9C9028B	8781442D	9C71D718	B64DD066	3AE19D25	D5031ADD
216	46CC50D9	7FF95368	F776E3B5	2E410804	CED9A695	1D866D7F	A320543A	24E5E0F1
224	AE8E4D49	44037FAE	3501EB60	37B74089	EDBC342D	AD175ACB	DCE2F54E	6355F75F
232	AEA504C7	F8D51064	2F2827B3	C1717A8E	BABA5F71	44A8880E	7A5035E2	4FE7872A
240	24A2AC56	D3D0FD67	8A9864C0	3F2287AD	A457A99F	78874FEF	3D696568	39025EF0
248	56907C90	2042CD0D	BA37BA2C	DEB0B269	0C53C865	B25F3B51	35AF5A64	9E72C5D2
256	8ECC8044	FE34C582	EBB15909	3BE488EC	5FE30F7C	98B800AF	4290309C	ED99771B
264	5091C145	1364496D	14CCAF5F	9E9275F2	22025FDE	0638B892	97C942F9	1F76466F
272	37C6E53B	C71C5238	140A1132	1D20D873	A26F110D	EF4E0D16	221F2307	0EC4BBCB
280	918C2EA8	3448DC3F	A717625F	7CDFA364	E31DC573	90401856	3B6BE1AF	0FFEEB7A
288	1261089F	3C242CF4	9EEF1C7D	105F2E05	DBF3EFE0	422C89EA	BAE3E757	8AEB6F62
296	0FCA4A6D	38FC88B0	C0735175	193C0DA4	EAF76B61	FB016C5A	37708E1C	A97D226A
304	57BE35E9	08F9812F	8084FB19	34D6B1C4	9408870C	D488D891	3CC31911	08C7C9BD
312	67372641	91D5E730	F9453046	0E44905B	40067A72	8F1A2169	36F52CA7	158E1636
320	83B4EFAF	C5611CB2	90AB8C26	50595477	54E9CE89	74D9A54D	AF6DF161	095F80CD
328	CAAB56DF	A8570496	902A0502	4DD5C32D	9E2ECE2C	546C9D3B	3E2F0ED1	63D674B5
336	FFE9317F	98D3842D	CD513834	AAF87FE8	C0641A37	EAA7D640	5F156DA4	F72EF8C2
344	FFE4A26C	5C84DF04	53BD4FAE	CE208A6C	AAA1AC32	D00F4CCD	C1DBD465	EA6AB45A
352	02DAA373	BEAF1B17	80C12B74	921A7B0F	13DB7E7A	6E369E05	491912AA	9115EB9A

All values are given in hexadecimal.

i	$rk[i]$	$rk[i+1]$	$rk[i+2]$	$rk[i+3]$	$rk[i+4]$	$rk[i+5]$	$rk[i+6]$	$rk[i+7]$
360	0824BCCB	8A082F2C	B744F5EA	27B2619D	94F225CE	9D295B43	6901C1EF	BF2A13CA
368	70B260B8	209C41B2	4F68E571	A562DBBC	55DE891D	5DC5FB79	E5CEE2EE	37012A43
376	61B2B567	96E1D827	29CA104C	2E6E8002	FF041885	4C88475D	0E1F205F	014813B4
384	943B7B54	97650B5B	AEAFAB76	6D1E638A	5F533927	6029BE5A	4851011E	052E90CE
392	9F41B790	24A7845A	DA5A9660	78E158BA	F4DB9B94	D5785A5D	6C2F5121	206BA45A
400	5415E4E2	FAC6D7D2	3789BDCB	51B94028	80A6D340	31EAAA58	C5A546B4	6314CEA1
408	9B7462B5	A16865EC	78735064	AEC85342	CEEEB2DD	892D01E9	6D0BEEFE	9A3C7101
416	95A2CE9C	D4779EFF	3BAD2E03	155C58B4	EAA740C5	8FCF004D	1D9C7919	93CD4C2A
424	5528BC8B	252D3F82	6F0C93A1	BEF1F763	07F85E96	5CF3EC81	89F6E685	6BFB94B3
432	3E037564	55CB367A	CA985384	A212A875	2347A2A9	1B4218C3	6A49431B	D9160080
440	F589BA2C	3F1E4E59	1A066A75	415183F2	BF4733CF	48A8D5FD	CED1D49F	6B75047C

All values are given in hexadecimal.

The value of h_1 is encrypted using the above $rk[\cdot]$ vector. We outline here the value of the internal state of the cipher in each round:

Round (i)	Left (L_i) Middle Rifht (B_i)	Middle Left (A_i) Right half (R_i)
0	F8EE1110 AA737414 7B6B98FF EE67A884	6DBFBD03 458B70EF 046D647D BBF660A9
	BD288D5F 88FD57AD DF4CF379 FE487408	AF710071 AD34BDFA EAE8CE3D F8738A73
	Input to left $F^4(\cdot)$:	6DBFBD03 458B70EF 046D647D BBF660A9
	Input to right $F^4(\cdot)$:	AF710071 AD34BDFA EAE8CE3D F8738A73
	Output from left $F^4(\cdot)$:	207D0ABA 222382D2 1941BABB 4DB521A2
	Output from right $F^4(\cdot)$:	5E8073B0 E0DE99DD 0D93DD22 1AA01CBA
1	6DBFBD03 458B70EF 046D647D BBF660A9	E3A8FEEF 6823CE70 D2DF2E5B E4E868B2
	AF710071 AD34BDFA EAE8CE3D F8738A73	D8931BAA 8850F6C6 622A2244 A3D28926
	Input to left $F^4(\cdot)$:	E3A8FEEF 6823CE70 D2DF2E5B E4E868B2
	Input to right $F^4(\cdot)$:	D8931BAA 8850F6C6 622A2244 A3D28926
	Output from left $F^4(\cdot)$:	7D8E85D2 DFCAC2F4 6A792242 ADB9F8A6
	Output from right $F^4(\cdot)$:	1382B6FB 0265BCBE 2F144864 312DF229
2	E3A8FEEF 6823CE70 D2DF2E5B E4E868B2	BCF3B68A AF510144 C5FC8659 C95E785A
	D8931BAA 8850F6C6 622A2244 A3D28926	103138D1 9A41B21B 6E14463F 164F980F
	Input to left $F^4(\cdot)$:	BCF3B68A AF510144 C5FC8659 C95E785A
	Input to right $F^4(\cdot)$:	103138D1 9A41B21B 6E14463F 164F980F
	Output from left $F^4(\cdot)$:	3D9EC20F 42A74F5D 47424EF3 BC80FE53
	Output from right $F^4(\cdot)$:	008E5B2E F0A0BF52 969381C9 1F79CA06
3	BCF3B68A AF510144 C5FC8659 C95E785A	D81D4084 78F04994 F4B9A38D BCAB4320
	103138D1 9A41B21B 6E14463F 164F980F	DE363CE0 2A84812D 959D60A8 586896E1
	Input to left $F^4(\cdot)$:	D81D4084 78F04994 F4B9A38D BCAB4320
	Input to right $F^4(\cdot)$:	DE363CE0 2A84812D 959D60A8 586896E1
	Output from left $F^4(\cdot)$:	5A0816BD 3D71EDD9 2DEA32EC 8FF96DD3
	Output from right $F^4(\cdot)$:	4CE4EC80 5E0D1696 FC82BB0D 596EBCD8
4	D81D4084 78F04994 F4B9A38D BCAB4320	5CD5D451 C44CA48D 9296FD32 4F2124D7
	DE363CE0 2A84812D 959D60A8 586896E1	E6FBA037 9220EC9D E816B4B5 46A71589
	Input to left $F^4(\cdot)$:	5CD5D451 C44CA48D 9296FD32 4F2124D7
	Input to right $F^4(\cdot)$:	E6FBA037 9220EC9D E816B4B5 46A71589
	Output from left $F^4(\cdot)$:	32587202 203ABC35 11EE02DC 7097BD44
	Output from right $F^4(\cdot)$:	C9BD8E27 BE043E0A 52C24964 E7679691
5	5CD5D451 C44CA48D 9296FD32 4F2124D7	178BB2C7 9480BF27 C75F29CC BF0F0070
	E6FBA037 9220EC9D E816B4B5 46A71589	EA453286 58CAF5A1 E557A151 CC3CFE64
	Input to left $F^4(\cdot)$:	178BB2C7 9480BF27 C75F29CC BF0F0070
	Input to right $F^4(\cdot)$:	EA453286 58CAF5A1 E557A151 CC3CFE64
	Output from left $F^4(\cdot)$:	BC08467C FAC48CD5 E22E8C67 4324CC7E
	Output from right $F^4(\cdot)$:	5E0D0800 D139FF91 0C5C55A1 4F0FBD02
6	178BB2C7 9480BF27 C75F29CC BF0F0070	B8F6A837 4319130C E44AE114 09A8A88B
	EA453286 58CAF5A1 E557A151 CC3CFE64	E0DD922D 3E882858 70B87155 0C05E8A9
	Input to left $F^4(\cdot)$:	B8F6A837 4319130C E44AE114 09A8A88B
	Input to right $F^4(\cdot)$:	E0DD922D 3E882858 70B87155 0C05E8A9
	Output from left $F^4(\cdot)$:	E4E3073A AAD09BDD 5EC7C14C 31AF5D7B
	Output from right $F^4(\cdot)$:	BF40381F B8D43CFD 6BD26536 68D618AD
7	B8F6A837 4319130C E44AE114 09A8A88B	55050A99 E01EC95C 8E85C467 A4EAE6C9
	E0DD922D 3E882858 70B87155 0C05E8A9	F368B5FD 3E5024FA 9998E880 8EA05D0B
	Input to left $F^4(\cdot)$:	55050A99 E01EC95C 8E85C467 A4EAE6C9
	Input to right $F^4(\cdot)$:	F368B5FD 3E5024FA 9998E880 8EA05D0B
	Output from left $F^4(\cdot)$:	156A3FFC 6AF4E8F1 ACDF60CB F081AC0E
	Output from right $F^4(\cdot)$:	043D55B9 0BC2BF80 0AE37313 DA2074DE

All values are given in hexadecimal.
The values are given before the round.

Round (i)	Left (L_i) Middle Rifht (B_i)	Middle Left (A_i) Right half (R_i)
8	55050A99 E01EC95C 8E85C467 A4EAE6C9	E4E0C794 354A97D8 7A5B0246 D6259C77
	F368B5FD 3E5024FA 9998E880 8EA05D0B	AD9C97CB 29EDFBFD 489581DF F9290485
	Input to left $F^4(\cdot)$:	E4E0C794 354A97D8 7A5B0246 D6259C77
	Input to right $F^4(\cdot)$:	AD9C97CB 29EDFBFD 489581DF F9290485
	Output from left $F^4(\cdot)$:	ED04A9F7 722E95CE 97556766 8E858627
	Output from right $F^4(\cdot)$:	BAB342A3 5AE3E11D 37CFD537 0C3AA863
9	E4E0C794 354A97D8 7A5B0246 D6259C77	49DBF75E 64B3C5E7 AE573DB7 829AF568
	AD9C97CB 29EDFBFD 489581DF F9290485	B801A36E 92305C92 19D0A301 2A6F60EE
	Input to left $F^4(\cdot)$:	49DBF75E 64B3C5E7 AE573DB7 829AF568
	Input to right $F^4(\cdot)$:	B801A36E 92305C92 19D0A301 2A6F60EE
	Output from left $F^4(\cdot)$:	7861FAAE E150B5E6 3EC934A5 A893D33F
	Output from right $F^4(\cdot)$:	6844897C 975ECF18 B68AC6B9 C55139C8
10	49DBF75E 64B3C5E7 AE573DB7 829AF568	C5D81EB7 BEB334E5 FE1F4766 3C783D4D
	B801A36E 92305C92 19D0A301 2A6F60EE	9C813D3A D41A223E 449236E3 7EB64F48
	Input to left $F^4(\cdot)$:	C5D81EB7 BEB334E5 FE1F4766 3C783D4D
	Input to right $F^4(\cdot)$:	9C813D3A D41A223E 449236E3 7EB64F48
	Output from left $F^4(\cdot)$:	4267C9D4 E83AAD3B D121F073 B1F1DA6A
	Output from right $F^4(\cdot)$:	B5DAB504 BD776389 005E7585 344EA958
11	C5D81EB7 BEB334E5 FE1F4766 3C783D4D	0DDB166A 2F473F1B 198ED684 1E21C9B6
	9C813D3A D41A223E 449236E3 7EB64F48	0BBC3E8A 8C8968DC 7F76CDC4 336B2F02
	Input to left $F^4(\cdot)$:	0DDB166A 2F473F1B 198ED684 1E21C9B6
	Input to right $F^4(\cdot)$:	0BBC3E8A 8C8968DC 7F76CDC4 336B2F02
	Output from left $F^4(\cdot)$:	887ADE10 EB4A74A0 F7C52BD1 B91D0C7A
	Output from right $F^4(\cdot)$:	636DC854 E046068C B4C3215E DD299AE8
12	0DDB166A 2F473F1B 198ED684 1E21C9B6	FFECF56E 345C24B2 F05117BD A39FD5A0
	0BBC3E8A 8C8968DC 7F76CDC4 336B2F02	4DA2C0A7 55F94045 09DA6CB7 85653137
	Input to left $F^4(\cdot)$:	FFECF56E 345C24B2 F05117BD A39FD5A0
	Input to right $F^4(\cdot)$:	4DA2C0A7 55F94045 09DA6CB7 85653137
	Output from left $F^4(\cdot)$:	40C1BA87 98D04B49 C2A6DBDA E4682125
	Output from right $F^4(\cdot)$:	DE37BFF6 45E10B5B 5ADA7F29 6FA8C847
13	FFECF56E 345C24B2 F05117BD A39FD5A0	D58B817C C9686387 25ACB2ED 5CC3E745
	4DA2C0A7 55F94045 09DA6CB7 85653137	4D1AACED B7977452 DB280D5E FA49E893
	Input to left $F^4(\cdot)$:	D58B817C C9686387 25ACB2ED 5CC3E745
	Input to right $F^4(\cdot)$:	4D1AACED B7977452 DB280D5E FA49E893
	Output from left $F^4(\cdot)$:	D04E190F AA5D5136 235398AD 69EF1A10
	Output from right $F^4(\cdot)$:	FC6E8046 B24597D9 59334145 280DFAA1
14	D58B817C C9686387 25ACB2ED 5CC3E745	B1CC40E1 E7BCD79C 50E92DF2 AD68CB96
	4D1AACED B7977452 DB280D5E FA49E893	2FA2EC61 9E017584 D3028F10 CA70CFB0

All values are given in hexadecimal.
The values are given before the round.

The ciphertext is XORed to the plaintext (in a Davies-Meyer mode), to produce the output of the compression function:

$h_1 =$ `F8EE1110 AA737414 7B6B98FF EE67A884 6DBFBD03 458B70EF 046D647D BBF660A9`
`BD288D5F 88FD57AD DF4CF379 FE487408 AF710071 AD34BDFA EAE8CE3D F8738A73`
$\oplus$
`D58B817C C9686387 25ACB2ED 5CC3E745 B1CC40E1 E7BCD79C 50E92DF2 AD68CB96`
`4D1AACED B7977452 DB280D5E FA49E893 2FA2EC61 9E017584 D3028F10 CA70CFB0`
$=$
$h_2 =$ `2D65906C 631B1793 5EC72A12 B2A44FC1 DC73FDE2 A237A773 5484498F 169EAB3F`
`F03221B2 3F6A23FF 0464FE27 04019C9B 80D3EC10 3335C87E 39EA412D 320345C3`

The digest is h_2:

$2D65906C$ $631B1793$ $5EC72A12$ $B2A44FC1$ $DC73FDE2$ $A237A773$ $5484498F$ $169EAB3F$
$F03221B2$ $3F6A23FF$ $0464FE27$ $04019C9B$ $80D3EC10$ $3335C87E$ $39EA412D$ $320345C3_x$