

The SHAvite-3 Hash Function

Tweaked Version

23.11.2009

Eli Biham^{1,*} and Orr Dunkelman^{2,3}

¹ Computer Science Department, Technion
Haifa 32000, Israel

`biham@cs.technion.ac.il`

² École Normale Supérieure
Département d'Informatique,
CNRS, INRIA

³ Faculty of Mathematics and Computer Science
Weizmann Institute of Science
P.O. Box 26
Rehovot 76100, Israel
`orr.dunkelman@weizmann.ac.il`

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 as a Feistel block cipher which iterates a round function based on the AES round function. 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 [30, 70] 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 first author was supported in part by the Israel MOD Research and Technology Unit.

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 $(2^{64} - 1)$ 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 for many platforms (both modern CPUs, as well as smart cards and 8-bit machines). Our current implementation of SHAvite-3 achieves for 256-bit digests a speed of 32.83 cycles per byte on a 32-bit machine and of 25.13 cycles per byte on a 64-bit machine. For 512-bit digests, SHAvite-3 achieves speeds of 55.90 cycles per byte on a 32-bit machine, and 35.86 cycles per byte on a 64-bit machine. As shown in [10], on future Intel CPUs which support the AES-NI instruction set, speeds of 5.6 and 5.5 cycles per byte, respectively, are attainable. These speeds suggest that on future platforms, SHAvite-3 is the fastest remaining candidate.

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 are outlined 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. We summarize the proposal in Section 9.

2 AES and Some Mathematical Background

Our construction relies on the round function used in AES [66]. 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 4×4 , 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 4×4 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 Federal Information Processing Standard 197 [66], using the subkey $subkey$ applied to the input x . Specifically,

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

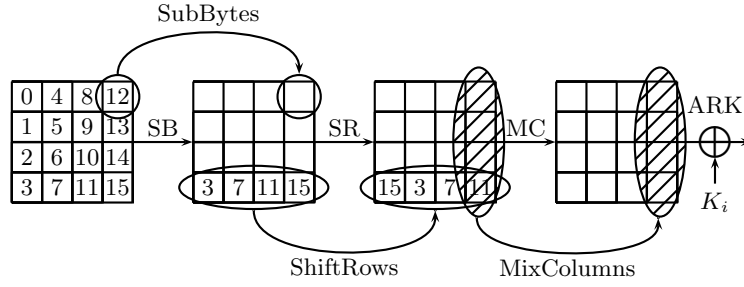


Fig. 1. An AES round

In AES, each byte represents a value in the field $GF(2^8)$, i.e., the byte value $0x13$ corresponds to the polynomial $x^4 + x + 1$. 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 Federal Information Processing Standard 197 [66]), first compute the product of $p(x)q(x)$ (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

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

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

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

– 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 [29, 55, 56]. 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, 32, 48]. Other undesired properties in such iteration were also suggested: extensions attacks (which lead to some differentiability results [26]), chosen target preimage attacks [47], and Trojan message attacks [2].

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 preserves the major security notions and is indeed second preimage resistant [23]. In particular, we show that the HAIFA mode of iteration is protected against the second preimage attacks, the chosen-target preimage attacks, and the Trojan message attacks of [1, 2, 32, 47, 48].

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 suggest 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 a 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 $salt$ 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 being signed), or even a counter. However, a careful application would ensure that the salt contains enough randomness to be unpredictable.

In order to compute $HAIFA_{salt}^C(M)$ using the compression function $C : \{0, 1\}^{m_c} \times \{0, 1\}^n \times \{0, 1\}^b \times \{0, 1\}^s \mapsto \{0, 1\}^{m_c}$ the message is first padded, and divided into ℓ blocks of n bits each, $pad(M) = M_1 || M_2 || \dots || M_\ell$. 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 $HAIFA_{salt}^C(M)$.

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 for prior invocations of the compression function.

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 the input parameters to the compression function including the salt, and tries to generate the same output. Under this strong assumption we assume that the adversary can even manipulate the *#bits* parameter.

Let M_1 and M_2 be the two colliding messages, i.e., $HAIFA_{salt_1}^C(M_1) = HAIFA_{salt_2}^C(M_2)$, 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 compression function calls are necessarily different. Therefore, a collision in the (full) hash function allows to find a collision in 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 $M_i^1 \neq M_i^2$ or $h_{i-1}^1 \neq h_{i-1}^2$ (where the superscript denotes the corresponding message), for which $C(h_{i-1}^1, M_i^1, salt, \#bits) = C(h_{i-1}^2, M_i^2, salt, \#bits)$, 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 this 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 the salt *salt* is known). 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 proves that HAIFA (when instantiated with a random oracle as a compression function) preserves the pseudorandom oracle property [26]. Hence, a HAIFA hash function can be distinguished after q queries to the compression function with probability at most $O(q^2/2^{m_c})$ (or if $m_c = m$ — with probability at most $O(q^2/2^m)$).

The prefix-free encoding also ensures the preservation of the pseudorandom function property of the compression function [7]. 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 $\min\{2^s, 2^m, 2^{m_c/2}\}$ 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 [32] is based on finding fix-points for the compression function, which can be iterated repeatedly. While it may be 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 the adversary to commit 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 [48] is based on constructing an expandable message using Joux’s multicollision technique [43]. As noted before, even if such a message was constructed, the cost of connecting it to the challenge message is equivalent to 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 $\ell - 1$ and find a message block that leads to a collision with a two message block starting at position $\ell - 2$) and find from this location a collision between one message block with three blocks. The result is a very limited “expandable message” of between two and five blocks which must be used starting at block $\ell - 2$ (and then its length is either 3 or 5 blocks) or at block $\ell - 1$ (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. We recall that despite the “expandable

message”, the connection phase still requires the adversary to commit to a specific location, i.e., the time complexity of the online phase of the attack is not reduced at all (and is still 2^m).

- **The Herding Attack (chosen target preimage attacks)** — In the herding attack [47], 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. If the salt length is equal to half the chaining value 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 tradeoff 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 very short “patches” to the message, and thus obtain a second preimage attack which is slightly slower than the one of [48], 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).
- **Trojan Message Attacks** — In this attack the adversary introduces a malicious suffix which allows efficiently finding second preimages to a restricted set of messages [2]. The two variants of the attack are using collisions in the compression function. Once the actual compression function is unknown in advance due to the salt, this attack is rendered impossible. By fixing the salt length to more than $m/2$, the attack becomes slower than second preimage attacks, even if the adversary is allowed to supply a different suffix for every possible salt.

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 [23] 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 [23]. 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 [63] 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 [38]. Thus, it 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 an 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 [38] 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-3₂₅₆ uses the compression function C_{256} and produces digests of up to 256 bits,
2. SHAvite-3₅₁₂ 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-3₂₅₆ (with truncation, as defined by HAIFA). The digest lengths 384 and 512 bits required by the call are to be produced by SHAvite-3₅₁₂.

4.1 Specifications of SHAvite-3₂₅₆

SHAvite-3₂₅₆ 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 $P[0, \dots, 7]$. The plaintext is divided into two halves $P = (L_0, R_0)$, where L_0 contains words $P[0, \dots, 3]$, and R_0 contains words $P[4, \dots, 7]$. 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:

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

$F^3(\cdot)$ 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):

$$F_{(k_i^0, k_i^1, k_i^2)}^3(x) = \text{AESRound}_{0^{128}}(\text{AESRound}_{k_i^2}(\text{AESRound}_{k_i^1}(x \oplus k_i^0))).$$

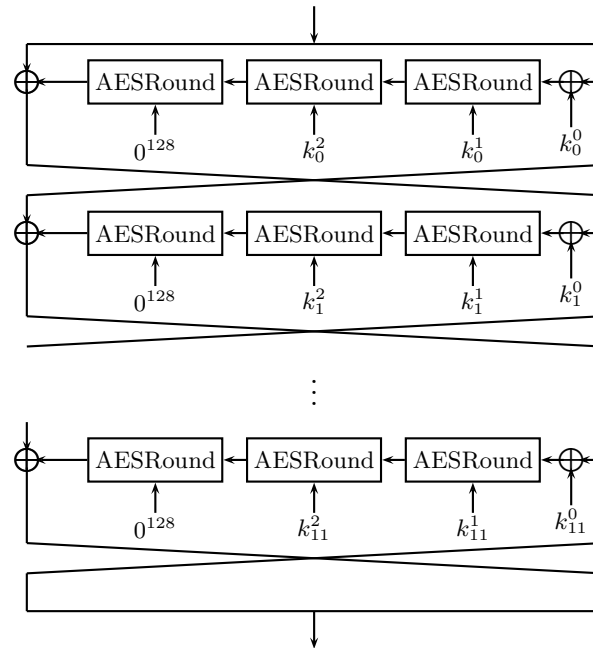


Fig. 2. The underlying block cipher of C_{256}

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 $rk[0, \dots, 143]$ be an array of 144 32-bit words, let $msg[0, \dots, 15]$ 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 $salt[0, \dots, 7]$ 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).

Eight of the produced words are XORed with the counter (four with $cnt[0]$ and four with $cnt[1]$), where in four of these positions, the counter is complemented,¹ thus preventing any slide properties of the cipher: $rk[16]$, $rk[58]^*$, $rk[87]^*$, and $rk[124]$ are XORed with $cnt[0]$ during their update, and $rk[17]^*$, $rk[57]$, $rk[86]$, and $rk[127]^*$, are XORed with $cnt[1]$. All the locations marked by $*$ are actually XORed with the complemented value of the respective counter.

A summary of the computation of rk is as follows:

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

$$t[0..3] = AESRound_{0^{128}} \left((rk[i-15] || rk[i-14] || rk[i-13] || rk[i-16]) \oplus (salt[0] || salt[1] || salt[2] || salt[3]) \right).$$

- (b) For $j = 0, \dots, 3$: $rk[i + j] \leftarrow t[j] \oplus rk[i + j - 4]$.
- (c) If $i = 16$ then $rk[16] \oplus = cnt[0]$ and $rk[17] \oplus = \overline{cnt[1]}$.
- (d) If $i = 56$ then $rk[57] \oplus = cnt[1]$ and $rk[58] \oplus = cnt[0]$.
- (e) $i \leftarrow i + 4$.
- (f) Let

$$t[0..3] = AESRound_{0^{128}} \left((rk[i-15] || rk[i-14] || rk[i-13] || rk[i-16]) \oplus (salt[4] || salt[5] || salt[6] || salt[7]) \right).$$

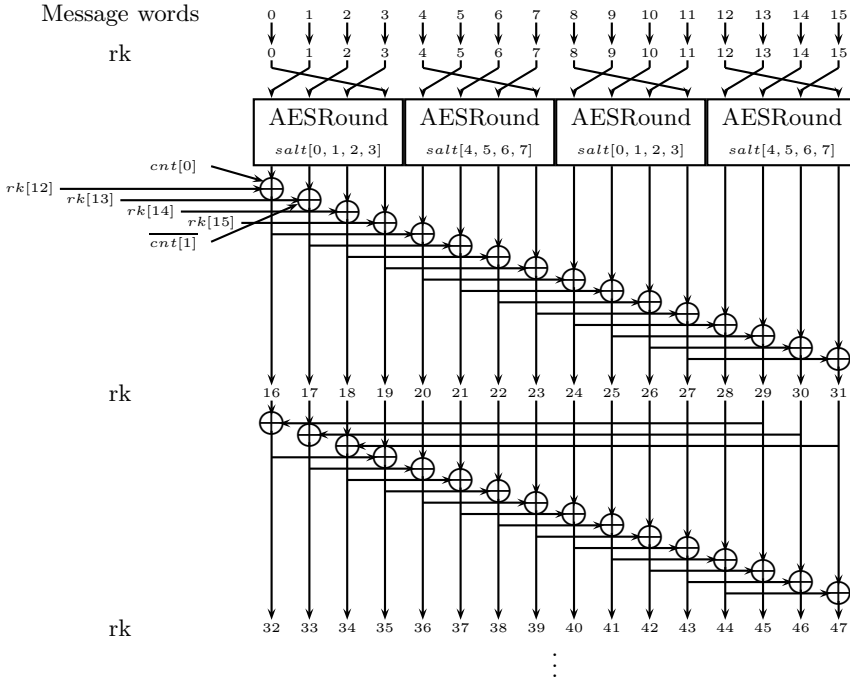
- (g) For $j = 0, \dots, 3$: $rk[i + j] \leftarrow t[j] \oplus rk[i + j - 4]$.
 - (h) If $i = 84$ then $rk[86] \oplus = cnt[1]$ and $rk[87] \oplus = \overline{cnt[0]}$.
 - (i) If $i = 124$ then $rk[124] \oplus = cnt[0]$ and $rk[127] \oplus = \overline{cnt[1]}$.
 - (j) $i \leftarrow i + 4$.
2. **Linear 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.,

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

¹ We note that this is the only difference between the original submitted version of SHAvite-3₂₅₆ and the tweaked version.



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

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

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,

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

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

$$MIV_{256} = C_{256}(0, 0, 0, 0) = \begin{matrix} \text{ACCF8F80} & \text{96589E66} & \text{43361E2F} & \text{ECC26B0F} \\ \text{D74AFFA7} & \text{FEDA3472} & \text{FAEC19EA} & \text{EE4C3008}_x. \end{matrix}$$

Table 2 lists the values of IV_m for 160, 224, and 256-bit versions of SHAvite-3₂₅₆.

IV_m	Value				
	$(IV_m[0] IV_m[1] \dots IV_m[7])$				
IV_{160}	E9E6C691	3DA59359	8FAEBDBB	1B797F0B	
	55712ABD	F639EE21	FDE027CE	8891EC2F _x	
IV_{224}	C4C67795	COB1817F	EAD88924	1ABB1BB0	
	E0C29152	BDE046BA	AEEECF99	58D509D8 _x	
IV_{256}	3EECF551	BF10819B	E6DC8559	F3E23FD5	
	431AEC73	79E3F731	98325F05	A92A31F1 _x	

Table 2. IV_m for Common Values of the Digest Size

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, $pad(M) = M_1||M_2||\dots||M_\ell$, and perform:

1. Set $\#bits \leftarrow 0$.
2. Set $h_0 \leftarrow IV_m$.
3. For $i = 1, \dots, \lfloor |M|/512 \rfloor$:
 - 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 $h_\ell = C_{256}(h_{\ell-1}, M_\ell, 0, salt)$ (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 $h_\ell = C_{256}(h_{\ell-1}, M_\ell, |M|, salt)$, else
 - Process some of the padding block with the last block containing message, i.e., compute $h_{\ell-1} = C_{256}(h_{\ell-2}, M_{\ell-1}, |M|, salt)$, and then compute $h_\ell = C_{256}(h_{\ell-1}, M_\ell, 0, salt)$ for processing the additional (partial) padding block.
5. Output $truncate_m(h_\ell)$, where $truncate_m(x)$ outputs the m leftmost bits of x , i.e., $x[0]||x[1]||\dots$

4.2 Specifications of SHAvite-3₅₁₂

SHAvite-3₅₁₂ 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 ($b = 128$).

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 nonlinear function $F^4(\cdot)$ is composed of four full rounds of AES.

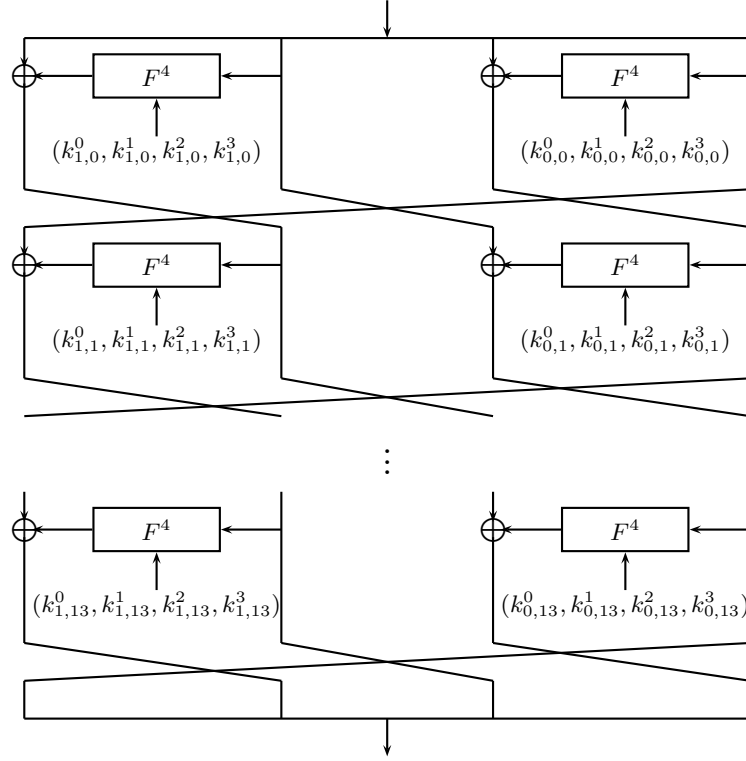


Fig. 4. The Underlying Block Cipher of C_{512}

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 $P[0, \dots, 15]$. The plaintext is divided into four 128-bit words $P = (L_0, A_0, B_0, R_0)$, where L_0 contains words $P[0, \dots, 3]$, and R_0 contains words $P[12, \dots, 15]$. 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:

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

$F^4(\cdot)$ accepts an input of 128 bits (either R_i or A_i) as well as a 512-bit subkey $RK_{i,j} = (k_{i,j}^0, k_{i,j}^1, k_{i,j}^2, k_{i,j}^3)$, and applies four rounds of AES, using k_i^0 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).

$$F_{(k_{i,j}^0, k_{i,j}^1, k_{i,j}^2, k_{i,j}^3)}^4(x) = \text{AESRound}_{0^{128}}(\text{AESRound}_{k_i^3}(\text{AESRound}_{k_i^2}(\text{AESRound}_{k_i^1}(x \oplus k_i^0)))).$$

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, a 128-bit counter, and a 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 $msg[0, \dots, 31]$ be the message array, $cnt[0, \dots, 3]$ be the counter array, and $salt[0, \dots, 15]$ 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 $cnt[i]$), where four times the XORed counter word is complemented.² The computation of $rk[\cdot]$ is done as follows:

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

$$t[0..3] = AESRound_{0^{128}} \left((rk[i-31] || rk[i-30] || rk[i-29] || rk[i-32]) \oplus (salt[0] || salt[1] || salt[2] || salt[3]) \right).$$

- (b) For $j = 0, \dots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
- (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

$$t[0..3] = AESRound_{0^{128}} \left((rk[i-31] || rk[i-30] || rk[i-29] || rk[i-32]) \oplus (salt[4] || salt[5] || salt[6] || salt[7]) \right).$$

- (f) For $j = 0, \dots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
- (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

$$t[0..3] = AESRound_{0^{128}} \left((rk[i-31] || rk[i-30] || rk[i-29] || rk[i-32]) \oplus (salt[8] || salt[9] || salt[10] || salt[11]) \right).$$

- (j) For $j = 0, \dots, 3$: $rk[i+j] \leftarrow t[j] \oplus rk[i-4+j]$.
- (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

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

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

² We note that this is the only difference between the original submitted version of SHAvite-3₅₁₂ and the tweaked version.

- (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.,

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

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

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

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

$$\begin{aligned}
 MIV_{512} = C_{512}(0, 0, 0, 0) &= \text{F0C11673} & \text{6F0196F5} & \text{ECCBD1EA} & \text{A6C59471} \\
 & \text{44AC3847} & \text{3BC53CB3} & \text{AD11AF21} & \text{4529BC86} \\
 & \text{0F2CFBA2} & \text{C73C31B7} & \text{B095486E} & \text{264A9D3F} \\
 & \text{73B16228} & \text{6FF3790A} & \text{E03124CD} & \text{F1320A9F}_x
 \end{aligned}$$

IV_m	Value ($IV_m[0] IV_m[1] \dots IV_m[15]$)				
IV_{384}	71F48510	A903A8AC	FE3216DD	0B2D2AD4	
	6672900A	41032819	15A7D780	B3CAB8D9	
	34EF4711	DE019FE8	4D674DC4	E056D96B	
	A35C016B	DD903BA7	8C1B09B4	2C3E9F25 _x	
IV_{512}	D5652B63	25F1E6EA	B18F48FA	A1EE3A47	
	C8B67B07	BDCE48D3	E3937B78	05DB5186	
	613BE326	A11FA303	90C833D4	79CEE316	
	1E1AF00F	2829B165	23B25F80	21E11499 _x	

Table 3. IV_m for Common Values of the Digest Size

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, $pad(M) = M_1||M_2||\dots||M_\ell$, and perform:

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

Table 3 lists the values of IV_m for 384-bit and 512-bit digests produced by SHAvite-3₅₁₂.

4.3 Changes from the Original Submission

As noted earlier, the only difference between the new C_{256} and its previous version is the fact that in four out of the eight positions where counter words are XORed, the counter word is complemented (specifically, twice $cnt[0]$ and twice $cnt[1]$). This insures that at least four of the eight counter words which enter the computation are non-zero.

The only difference between the new C_{512} and its previous version is the fact that four out of sixteen positions where the counter words are XORed, the counter word is complemented (each of the four words is complemented once). This insures that at least four of the sixteen counter words which enter the computation are non-zero.

4.4 Degenerate Salts

In applications where a salt cannot be used or when the additional functionality is not needed (possibly in exchange for a 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, diffused 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 can be implemented 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 demonstrated by the recent results, this is not sufficient, as the adversary can control the key (through the 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 have as low differential probability as possible (it may be at most $2^{-m/2}$ for an 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 (as happened in the recent attacks on AES [19–21]), 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.
- Enveloped Merkle-Damgård — While the enveloped Merkle-Damgård mode offers the preservation of the pseudo random properties of the compression function, it does not offer full second preimage resistance for long messages and is not secure against the herding attack. Hence, we decided to avoid the use of this mode.
- 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 [2].³ This lack of security, as well as the fact that the memory requirements for 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 [12], 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, which is a very dangerous practice [28, 60].
- Widepipe — This mode offers security with a relatively small performance penalty [52]. 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.

³ The attack of [2] is applicable whenever all compression functions used in the tree are the same. This is not true when there is an additional input to the compression function which changes in different calls, independent of the message.

The salt must be at least half the size of the chaining value in order to protect against herding attacks and Trojan message attacks (i.e., $s > m_c/2$) [17]. 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, as well as when the counter is complemented.

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 be easily 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., $\sqrt{2}, \phi, \dots$).

As for the tap positions, we first note that we use a register of 16 32-bit words in SHAvite-3₂₅₆ and 32 words in SHAvite-3₅₁₂ 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 the nonlinear expansion step.

In SHAvite-3₂₅₆, 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-3₅₁₂.

For the locations where the counter is mixed, we chose locations which allow good mixing of the counter, and prevent slide properties. For SHAvite-3₂₅₆ 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-3₅₁₂ 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.

Following the concerns raised in [57, 61] (which had no impact on the security of SHAvite-3 as a hash function), we tweaked the compression functions of SHAvite-3. The tweak was chosen to be the complementation of counter words in four locations (out of the eight or 16 locations where the counter is XORed into the expanded message). The complementation locations were chosen to ensure that no slide properties would exist even for specially chosen message and salt values. For ease of implementation, each time the counter is XORed into the state, the last word of the counter is complemented. This allows software implementations to precompute the complementation, thus reducing the effect this may have on the running time.

Hence, there are no slide properties in the underlying block ciphers. Even if (somehow) the attacker can generate 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 arguments 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 potentially weak values is solved by AES' constants and its 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 any additional round constants.

6 The Security of SHA_{vite-3}

The security of SHA_{vite-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

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

Lemma 1. ([46]) *The exact 2-round AES maximal expected differential probability is equal to $53/2^{34} \approx 1.656 \cdot 2^{-29}$.*

Lemma 2. ([46]) *The 4-round AES maximal expected differential probability is upper bounded by $(53/2^{34})^4 \approx 1.881 \cdot 2^{-114}$.*

Note that 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 [59].

As C_{256} uses 3-round AES as the round function, we use the following lemma:

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 four 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 output difference of the 2-round differentials in the first two rounds. In the computation of the upper bound, we take into consideration the fact that any 2-round differential has probability of 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} that leads to the desired output difference. Therefore, we conclude that the maximal expected differential probability is upper bounded by

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

□

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, and 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,⁴ Hence, similarly to the case of E^{256} , there is no 2-round iterative differential characteristic.
- Recall that $F^4(\cdot)$ is composed of four 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. Due to the bijectiveness of the round function. Hence, there are at least two active rounds, and the maximal differential probability of any 3-round differential characteristics is at most $(2^{-113})^2 = 2^{-226}$.
- 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 the use of linear cryptanalysis 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, suggests 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 [44].

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 their 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 [46] 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 (following the structural impossible differential suggested in [49]). The strong round function suggests that there are no longer impossible differentials, and we conclude that E^{512} is secure against impossible differential cryptanalysis as well.

⁴ 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 and final permutations of DES.

- **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 (see [24, 51]), 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. We also note that this seems to render cube attacks [33] on the full cipher unsuccessful.
- **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 SHA_{vite-3}. Despite that, SHA_{vite-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. As observed in [57, 61], without the complementation of the counter, the case $\#bits = 0$ (which happens only during initializations and the processing of a full padding block, i.e., where the adversary has almost no control over the inputs) the block ciphers possess some slid properties and fix points (for message block $m = 0$ and salt $salt = 52\ 52 \dots 52_x$). In [22] several other special relationships for $\#bits = 0$, $m = 0$, and $salt = 52\ 52 \dots 52_x$ are discussed. Again, this issue is solved by the tweak.
- **Square Attacks** — SHA_{vite-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} . 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 the output in any layer of AES rounds in the message expansion of SHA_{vite-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 adversary 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 linearly 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 adversary makes 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-3₂₅₆ and 512 more bits of freedom in SHAvite-3₅₁₂), 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 [3] which claim that HAIFA does not preserve various properties of the compression function. The claim of [3] is based on constructing a special compression function which possesses 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 possess the weaknesses used in the special construction (or any weakness for that matter). Moreover, recent results obtained in [23], 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 or time-memory tradeoff attack).

Another security property that HAIFA hash functions (and thus SHAvite-3) possess 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 signature schemes such as RSA-PSS and message authentication codes such as 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) [65], one could use the randomness as the salt as well (or as part of the salt). As the use of salts increases the security of the hash function (just like in randomized hashing [38]), we suggest that new signature schemes support 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 [38]. 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 [68].

In [38] 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 [6]. 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

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 manner⁵ 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:

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

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 's 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 a related-key pseudorandom function). As SHAvite-3 has a

⁵ 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 (where the salt does not enter the real compression function) are identified and discussed in [5].

Construction	Null string	1500-Byte String	n -Byte String
SHA-256	1	24	$\lceil (n+8)/64 \rceil$
HMAC-SHA-256	2	25	$1 + \lceil (n+8)/64 \rceil$
SHAvite-3	1	24	$\lceil (n+10)/64 \rceil$
SHAvite-3-MAC	1	24	$\lceil (n+10)/64 \rceil$

Table 4. Number of Compression Function Calls of SHA-256, HMAC-SHA-256, SHAvite-3, and SHAvite-3-MAC (for 256-bit digest/tag)

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-3₂₅₆ as the hash function, while for tags of longer tags (of up to 512 bits), one should use SHAvite-3₅₁₂. Of course, in this case the key used as salt is to be kept secret and never used publicly.

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. In Table 4 we compare the number of compression function calls when using SHA-256, HMAC-SHA-256, SHAvite-3, and SHAvite-3-MAC (when they are used to produce a 256-bit digest/tag).

The performance advantage may seem small (one compression function call), but for short messages (up to 53 bytes), it offers a 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. Moreover, it seems that on Intel CPUs with the new AES-NI instruction, SHAvite-3 is going to be the fastest candidate [10].

The running times of our current (slightly optimized) ANSI-C code is 32.83 cycles per byte for 224-/256-bit digests, and 55.90 cycles per byte for 384-/512-bit digests on 32-bit Intel machines. On a 64-bit machine, the corresponding running times are 25.13 and 35.86 cycles per 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 last quarter of 2009 or the first quarter of 2010) and to the AMD CPUs (expected in 2011), these speeds will

improve significantly, as instead of performing an AES round in 21–29 cycles (the best known speeds at the moment on common CPUs), the speed of an AES round would be reduced to roughly 6 cycles and several of these rounds can run in parallel. 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. As shown in [10], the actual gains for SHAvite-3₂₅₆ is expected to be about 77%, and for SHAvite-3₅₁₂ is expected to be about 84%.

Finally, we note that the tweak is not expected to invalidate most of the previous work done on measuring and implementing SHAvite-3. This follows the fact that the tweak is composed of negating four words, which can be done in software in four 32-bit operations or in very few hardware gates.

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 [27]:

“The only field multiplication used in this algorithm is multiplication with the element 02, denoted by ‘xtime’.

$$\begin{aligned} t &= a[0] \oplus a[1] \oplus a[2] \oplus a[3]; & / * a \text{ is a column} * / \\ u &= a[0]; \\ v &= a[0] \oplus a[1]; & v = \text{xtime}(v); & a[0] = a[0] \oplus v \oplus t; \\ v &= a[1] \oplus a[2]; & v = \text{xtime}(v); & a[1] = a[1] \oplus v \oplus t; \\ v &= a[2] \oplus a[3]; & v = \text{xtime}(v); & a[2] = a[2] \oplus v \oplus t; \\ v &= a[3] \oplus u; & v = \text{xtime}(v); & a[3] = a[3] \oplus v \oplus t; \end{aligned}$$

SHAvite-3₂₅₆ 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 [62] an AES-128 implementation which takes 3766 cycles for a block is reported on an AVR processor. This speed is about 377 cycles per full AES round. Hence, the running time of SHAvite-3₂₅₆ on AVR is expected to be about 20370 cycles per each invocation of the compression function, or about 318 cycles per byte. SHAvite-3₅₁₂ 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 [64] 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 [4, 11] 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 32.83 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, compiled with gcc 4.4.1).

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) [11]. 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-3₅₁₂ has a running time of 55.90 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 32.83 cycles per byte for (not-well-optimized) SHAvite-3₂₅₆, the fastest SHA-1 implementation (obtained from the OpenSSL library) has a running time of 9.38 cycles per byte, SHA-256 had a running time of 27.29 cycles per byte, and SHA-512 had a running time of 78.38 cycles per byte. All measurements were done using the NESSIE test suite [58] using code from OpenSSL and internal NESSIE code.

In the eBASH project [35], the timings of various SHA-3 candidates (as well as other hash functions) on various platforms were measured. The measurements for SHAvite-3 were done on one variant of the code which we provided, and that was optimized to a specific type of a 32-bit machine without salt support (i.e., the fixed salt was used). The running times on x86 platforms (Intel and AMD) varies between 28.73 and 84.42 cycles/byte for SHAvite-3₂₅₆ (the lower speeds are usually obtained on older machines or older compilers or both), validating our previous speed claims. For PowerPCs with 32-bit CPUs the speeds of SHAvite-3₂₅₆ varied between 20.62 and 43.99 cycles per byte. For SHAvite-3₅₁₂ the measured speeds on x86 platforms vary between 55.30 and 242.09 cycles per byte. For PowerPCs with 32-bit CPUs, the running times are between 32.00 and 184.78 cycles per byte.

8.1.3 64-Bit Machines There are several approaches for implementing 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 instructions available on newer machines. A different approach is the use of bit-sliced implementation proposed in [13], which were applied in [45, 53, 54] 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 machines can reach 10.5 cycles per byte in counter mode, even without bit-slicing [11]. 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-3₅₁₂ reveals prospective speed of about 28.4–31.8 cycles per byte in an optimized implementation.

Hash Function	32-Bit Platform	64-Bit Platform
MD5	5.67	6.96
SHA-1	9.38	7.34
SHA-256	27.29	19.08
SHA-512	78.38	14.71
SHAvite-3 ₂₅₆ (measured)	32.83	25.13
SHAvite-3 ₂₅₆ (eBASH, Intel/AMD)	28.73–84.42	22.79–61.24
SHAvite-3 ₂₅₆ (eBASH, PowerPC)	20.62–43.99	24.71–39.64
SHAvite-3 ₂₅₆ (conjectured)	26.6	18.6
SHAvite-3 ₂₅₆ (with AES inst.)		5.6 [10]
SHAvite-3 ₅₁₂ (measured)	55.90	35.86
SHAvite-3 ₅₁₂ (eBASH, Intel/AMD)	55.30–242.09	40.28–255.10
SHAvite-3 ₅₁₂ (eBASH, PowerPC)	32.00–184.78	38.41–64.39
SHAvite-3 ₅₁₂ (conjectured)	35.3	28.4
SHAvite-3 ₅₁₂ (with AES inst.)		5.5 [10]

Table 5. Speed Comparison of Hash Functions (in cycles/byte)

At the moment, our C-implementation of SHAvite-3₂₅₆ has a running time of 25.13 cycles per byte on AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ (2200 MHz, 512 KB cache, 1 GB RAM, compiled with gcc 4.4.1). The code of SHAvite-3₅₁₂ has a running time of 35.86 cycles per byte. For comparison, on this machine, SHA-1 takes 7.34 cycles per byte, SHA-256 takes 19.08 cycles per byte, and SHA-512 takes 14.71 cycles per byte.

The measurements of eBASH for 64-bit Intel and AMD platforms were between 22.79 and 61.24 cycles per byte for SHAvite-3₂₅₆ and between 24.71 and 39.64 cycles per byte for PowerPC platforms. For SHAvite-3₅₁₂ the measurements on Intel and AMD platforms were 40.28 and 255.10 cycles per byte. For PowerPCs, the corresponding range is 38.41 to 64.39 cycles per byte.

8.2 Future Platforms

It is evident that adding the set of AES instructions to Intel and AMD CPUs is expected to speed up AES implementations as well as SHAvite-3's implementations. The expected latency of the new Intel instruction is 6, i.e., it would take 6 cycles to perform one AES round [42]. Also, the platform is expected to allow multiple calls for the instruction (i.e., it is possible to compute two independent AES rounds in parallel within 7 cycles).

With such an instruction, and sufficient number of registers, we expect that the speed of SHAvite-3₂₅₆ 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. In [10] the untweaked version of SHAvite-3 was timed, and the outcome was 5.6 cycles/byte with the new AES-NI extension. We note that of all the second round candidates considered in [10], SHAvite-3 was the fastest. As the tweak is expected to have little effect on the time measures, we can safely estimate the running time of the tweaked version as having the same value, i.e., 5.6 cycles/byte.

For SHAvite-3₅₁₂, where we expect even better use of the instruction, non-interleaved code (with 168 AES rounds and 528 32-bit XORs) is expected to have a running time of about 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 as was shown in [10]. For the untweaked SHAvite-3₅₁₂ the running time was measured to be 5.5 cycles/byte with the new

AES-NI extension. As before, the tweak is not expected to affect the performance, and thus, we expect a similar speed for the tweaked version.

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. At the moment, we discuss two 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 [39]. The suggested implementation uses about 3100 gates, and has a throughput of 121 Mbps in a 152 MHz maximal frequency (the implementation uses a 0.13μ technology). Of the 3100 gates, about 60% (i.e., 1860 gates) are reported to store 256 bits of the internal state (a rate of about 7.2 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 applications 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 [41] 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 [37], 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 [74]. 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-3₂₅₆ in Hardware When implementing SHAvite-3₂₅₆ 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 32-bit 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, an implementation based on [39] would need another 128 bits 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 1240 gates for the AES core,⁶ along with about 100 more gates for the XORs and control overhead. This results in a full implementation of SHAvite-3₂₅₆ in about 10100 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 16 cycles for an AES round (at 152 MHz), which implies a speed of about 840 cycles for an invocation of C_{256} , or a throughput of about 93.5 Mbps at 153 MHz clock rate.

For the fast ASIC implementation we consider a similar methodology, but using the different implementation figures of [41]. First of all, we note that SHAvite-3₂₅₆ 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 two AES round implementations (22,850 gates each), 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 critical datapath of C_{256} is of 36 rounds of AES, and thus, it seems that one invocation of this implementation 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-3₂₅₆. 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 there 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).

⁶ 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.

8.3.3 Implementing SHAvite-3₅₁₂ in Hardware Applying the same methodology as for SHAvite-3₂₅₆ 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 18400 gates, and to achieve a speed of about 57.9 Mbps.

When implementing C_{512} targeting a fast implementation in ASIC using the methodology described in [41], three AES round cores need to be used with some additional memory.⁷ This increases the circuit size to about 81,000 gates. As the critical 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 1.12 Gbps).

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 Indestege, Nathan Keller, Gaëtan Leurent, Osnat Ornan, Ron Rivest, Adi Shamir, and Frederik Vercauteren. Our discussions with them, as well as their good advice and heritage, made SHAvite-3 a better hash function.

We also thank Thomas Peyrin, Mridul Nandi, and Souradyuti Paul for their analysis of the security of SHAvite-3. We would also wish to thank Ryad Benadjila, Olivier Billet, Shay Gueron, Matthew J.B. Robshaw for implementing SHAvite-3 with the new AES-NI instruction set. The comments of Wild Alexander, Thomas Plos, and Wang Xueyang who pointed out a small typo in the reference implementation of SHAvite-3 are also appreciated.

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, Charles Bouillaguet, Orr Dunkelman, John Kelsey, *Herding, Second Preimage and Trojan Message Attacks Beyond Merkle-Damgård*, presented at Selected Areas in Cryptography 2009.
3. 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.
4. Kazumaro Aoki, Helger Lipmaa, *Fast Implementations of AES candidates*, proceedings of the third AES conference, pp. 106–120, New York, 2000.

⁷ The three cores are used as follows: one in each $F^4(\cdot)$, and one for the message expansion. There is a requirement for some additional memory in the message expansion in this approach.

5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Ryad Benadjila, Olivier Billet, Shay Gueron, Matthew J.B. Robshaw, *The Intel AES Instructions Set and the SHA-3 Candidates*, accepted to ASIACRYPT 2009, available online at <http://crypto.rd.francetelecom.com/sha3/AES/paper/>.
11. Daniel J. Bernstein, Peter Schwabe, *New AES Software Speed Records*, proceedings of INDOCRYPT 2008, Lecture Notes in Computer Science 5365, pp. 322–336, Springer-Verlag, 2008.
12. 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.
13. 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.
14. 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.
15. 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.
16. Eli Biham, Orr Dunkelman, *A Framework for Iterative Hash Functions — HAIFA*, NIST 2nd hash function workshop, Santa Barbara, August 2006.
17. Eli Biham, Orr Dunkelman, *A Framework for Iterative Hash Functions — HAIFA*, IACR ePrint report 2007/278.
18. Eli Biham, Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.
19. Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, Adi Shamir, *Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds*, IACR ePrint report 2009/374.
20. Alex Biryukov, Dmitry Khovratovich, Ivica Nikolic, *Distinguisher and Related-Key Attack on the Full AES-256*, Advances in Cryptology, proceedings of CRYPTO 2009, Lecture Notes in Computer Science 5677, pp. 231–249, Springer-Verlag, 2009.
21. Alex Biryukov, Dmitry Khovratovich, *Related-key Cryptanalysis of the Full AES-192 and AES-256*, accepted to ASIACRYPT 2009, available online at <http://eprint.iacr.org/2009/317.pdf>.
22. Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Gaëtan Leurent, *New Self-Similarity Attack*, preprint, September 2009.
23. Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Sébastien Zimmer, *(Revisiting the) Second Preimage Resistance of Some Iterated Hash Functions*, preprint, 2008.
24. Carlos Cid, Gaëtan Leurent, *An Analysis of the XSL Algorithm*, Advances in Cryptology, proceedings of ASIACRYPT 2005, Lecture Notes in Computer Science 3788, pp. 333–352, Springer-Verlag, 2005.
25. 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.

26. 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.
27. Joan Daemen, Vincent Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*, Springer-Verlag, 2002.
28. 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.
29. 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.
30. 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.
31. 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.
32. Richard D. Dean, *Formal Aspects of Mobile Code Security*, Ph.D. dissertation, Princeton University, 1999.
33. Itai Dinur, Adi Shamir, *Cube Attacks on Tweakable Black Box Polynomials*, Advances in Cryptology, proceedings of EUROCRYPT 2009, Lecture Notes in Computer Science 5479, pp. 278–299, Springer-Verlag, 2009.
34. ECRYPT, *State of the Art in Hardware Architectures*, report D.VAM.2, September 2005, available online at <http://www.ecrypt.eu.org/documents.html>.
35. ECRYPT, *ECRYPT Benchmarking of All Submitted Hashes*, available online at <http://bench.cr.yt.jp/results-hash.html>.
36. 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.
37. 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.
38. Shai Halevi, 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.
39. Panu Hämäläinen, Timo Alho, Marko Hännikäinen, Timo D. Hämäläinen, *Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core*, Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools, IEEE Computer Society, 2006.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. Emilia Käyser, Peter Schwabe, *Faster and Timing-Attack Resistant AES-GCM*, proceedings of Cryptographic Hardware and Embedded Systems — CHES 2009, Lecture Notes in Computer Science 5747, pp. 1–17, Springer-Verlag, 2009.

46. Liam Keliher, Jiayuan Sui, *Exact Maximum Expected Differential and Linear Probability for 2-Round Advanced Encryption Standard (AES)*, IACR ePrint report 2005/321, 2005.
47. John Kelsey, Tadayoshi Kohno, *Herdling 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.
48. 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.
49. 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.
50. Lars R. Knudsen, Vincent Rijmen, *Known-Key Distinguishers for Some Block Ciphers*, Advances in Cryptology, proceedings of ASIACRYPT 2007, Lecture Notes in Computer Science 4833, pp. 315–324, Springer-Verlag, 2007.
51. Chu-Wee Lim, Khoongming Khoo, *An Analysis of XSL Applied to BES*, proceedings of Fast Software Encryption 2007, Lecture Notes in Computer Science 4593, pp. 242–253, Springer-Verlag, 2007.
52. 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.
53. 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.
54. 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.
55. Ralph C. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research press, 1982.
56. 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.
57. Mridul Nandi, Souradyuti Paul, *OFFICIAL COMMENT: SHAvite-3*, 2009, available online at <http://ehash.iaik.tugraz.at/uploads/5/5c/NandiP-SHAvite-3.txt>.
58. NESSIE, *The NESSIE Test Suite*, version 3.1.1, 2002.
59. 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.
60. Thomas Peyrin, *Cryptanalysis of Grindahl*, Advances in Cryptology, proceedings of ASIACRYPT 2007, Lecture Notes in Computer Science 4833, pp. 551–567, Springer-Verlag, 2007.
61. Thomas Peyrin, *Chosen-salt, chosen-counter, pseudo-collision on SHAvite-3 compression function*, 2009, available online at <http://ehash.iaik.tugraz.at/uploads/e/ea/Peyrin-SHAvite-3.txt>.
62. 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.
63. 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.
64. 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.
65. US National Institute of Standards and Technology, *Digital Signature Standard*, Federal Information Processing Standards Publications No. 186-2, 2000.

66. US National Institute of Standards and Technology, *Advanced Encryption Standard*, Federal Information Processing Standards Publications No. 197, 2001.
67. US National Institute of Standards and Technology, *Secure Hash Standard*, Federal Information Processing Standards Publications No. 180-2, 2002.
68. US National Institute of Standards and Technology, *Randomized Hashing for Digital Signatures*, Draft NIST Special Publication 800-106, 2008.
69. 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.
70. 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.
71. 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.
72. 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.
73. Gideon Yuval, *How to Swindle Rabin*, Cryptologia, Vol. 3, pp. 187–190, 1979.
74. 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))
“”	2A86D92E E32FA083 F4CFF0E5 4CCCA99C 4596A790 978E0A36 A78143A4 _x
“A”	882B05AF 4A011380 4000F5D3 A0DC3478 FDD0A39C E9CF122F 693DAA65 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	ODDD788 B73DCA82 A2A29B3F 5BC42C8C 7FDEA1A1 9F488808 EB430291 _x
“AAA...AA” (1,000,000 times)	27851E6D F3835375 67D6FF92 06DA6F82 77E54A48 FB9EC067 9FC7B531 _x

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

Message (M)	Digest (SHAvite-3 _{11111111 ... 11111111_x} (M))
“”	FD92F687 245D0A4C F6767075 F79AAFA4 85BBB828 368CC6CE BF35A7CE _x
“A”	339E7CCB C2981B9D 74BB92C0 A0107783 97CFDEF7 0B00469E DEF2CB97 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	EDD13EC8 D7F46702 B1DB7FA9 1BC4A20E 7BDEF3ED E7C5EE7C 199B3CD8 _x
“AAA... AA” (1,000,000 times)	429B1D9B 832771FA 9EC815A4 708FEFAD A47D115D A572320B 0DC30C2F _x

A.2 Digests of 256-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-3 ₀ (M))
“”	F7A5E74A 15262D71 F90BE000 11B7B2D7 E7779315 94259872 03B260B3 FA7DA608 _x
“A”	E27E9117 438183FB 44ACC3AA 9E036239 2386517B 375A196C DOB810E9 2724A94D _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	7FD46C27 CBDEF418 2C1F57C7 4FD8C6B9 AE53DC95 1C49D4BF E57685AD D4675EFC _x
“AAA... AA” (1,000,000 times)	913766F3 6FC9F8C9 893DE567 501C3A43 C261FCE9 E8E29FF8 DB289A64 74E64413 _x

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

Message (M)	Digest (SHAvite-3 _{22222222 ... 22222222_x} (M))
“”	7ED3E8FE 360D7085 58A1217C 50E9CB46 BF1EC62F 1FFEA7F9 CF3342C2 2FDFBE1E _x
“A”	605F0589 4B37265A 9E82ABB5 47E09FDD 5E99BD2C 829F073D 4250EDD2 890552F4 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	04D07FE8 414EE35C 11B559B9 8D8BB4C2 EDE5EFAC 39A55AF1 80626E6E B5E80ADF _x
“AAA... AA” (1,000,000 times)	75DCBF98 54105E48 E10A5F1A CFB1AA5F DED2CF0B 47F6798A 25238315 D4F9C001 _x

A.3 Digests of 384-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-3 ₀ (M))
“”	88845B9F FAC2C991 22FE1ACC C9C89EE6 2DOA9911 D9CA5343 22FAFB5E 7E93C3D2 95661762 A8C001C6 90DEB676 7E3519DA _x
“A”	42C5AF7D 083A0490 C18813D9 27611AF4 746C9953 6A235E30 82CD3C31 529A1BE3 12E9976C 5D3DC612 57111B7D AD892E22 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	81C569B2 5D641DF5 B1A257EB C5BE29C7 F60A30D8 F7120DBF 371191E4 9EAD7515 735F2685 748CF263 418F6090 9A730FA1 _x
“AAA...AA” (1,000,000 times)	1AAE3C75 7BEC5022 2C856BCC 4F3A71B9 EF147DB7 3CA8A80E CD081294 88FCE133 477D2B9E 8E33DE6B 465D746B A3D7C257 _x

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

Message (M)	Digest (SHAvite-3 _{33333333...33333333_x} (M))
“”	152990BB 7046D2D3 3A5DB501 550FC09F 50BD2C95 4FCA7B7C 6DDC5A77 0BAFB46B 881FA20A 2336C0DD B2880026 8DB8242D _x
“A”	5B56C65C 508F6258 3955BA4F 0A05F37C 2759E4C8 4308DDCB 526585C4 679AC9C3 B004CCE2 CC55392B 9D5040A3 CD4BB266 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	8A5821B0 1359B5E8 17BD90F8 6503C5D3 EF536A89 B5F82CC7 40AB94F8 06378796 45DBFOCE 98C1906C 49BFEA31 D095AC58 _x
“AAA...AA” (1,000,000 times)	C5590E4A C158386F 652F4F9C C79AB271 4C833E54 1BD96505 EBA3AB9D E6EC9667 7B030E10 30937BDB 4A91D581 0B598A18 _x

A.4 Digests of 512-Bit Long

For $salt = 0$:

Message (M)	Digest (SHAvite-3 ₀ (M))
“”	DCEFAEBD EBD91BF5 12496E73 C6B1C634 C79A8350 F7DA4176 BB78619E 723E3B28 83BF52B0 650F0F4E 3DE1005C EADCEE8E 88E428F8 280EE5A8 204E22AB 440F9547 _x
“A”	7101935E 7A7E833B 8AD272B2 A3D63706 2D95FA5F FE568254 D47588A8 2586CC9C D800E2D4 D436FA84 D213894A 79CEE4FE 8C564096 05616842 3F478714 CD40B0D4 _x
“ABCDEFGHJKLMNOPQRSTUVWXYZ”	5BBA098A 1334EC66 AD82CDC3 EDF838EC 7C32AF0B 62A27329 4F8E086C E91266E2 C664B910 7CED06F2 05DD28AC 9F55BEA3 4D8C323A D9F18C8A 86476216 0F701C22 _x
“AAA...AA” (1,000,000 times)	83D48B76 194F854B 6961A072 5254D22F 5C0A90AC EDFD064E 752D8455 059C1392 F3C8F3DA 5DD11DOC 388AF702 OD3083CC 4C214AC2 4745892D DDEC94DA 54FA93DF _x

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

Message (M)	Digest (SHAvite-3 $_{44444444\ \dots\ 44444444_x}(M)$)
“”	BF41E32F FDCD6515 3FEAE603 5FCA1C85 72508A6B C97A1A4E C510DF1A CFA77E74 8293D797 FE34E135 C340DDEB 8A2D7ED8 C38E8482 39434A2E 8FD60DE0 B587E66D _x
“A”	8C4EACD0 C6D4C409 CABA8841 BBCCD07B 88389F45 133494CE 7717BAD7 957B7085 467355A2 440DBDD3 D385019C C0319733 1E073D98 82A558D4 9AE6CA62 0B168A34 _x
“ABCDEFGHIJKLMNOPQRSTUVWXYZ”	4E88F55B 8AA95A99 1B07880C 8D1B303A DB9902AF E0837D3E D7BF7152 071B8402 0DCD4246 57F17300 8E8C63E7 7294FA75 913E2712 55C1A9BE FF268E96 A4551B54 _x
“AAA...AA” (1,000,000 times)	C33374FF D173A1D7 69090351 F2BF8B8F 67B3105F 7618E140 782648C9 2BF3416B 90030843 234FB575 976F26C6 B31B3E72 51363954 A43D3AF9 59D7702C 5519105A _x