Vortex: A New Family of One Way Hash Functions based on Rijndael Rounds and Carry-less Multiplication

Shay Gueron and Michael E. Kounavis
Intel Corporation, October, 2008

Supporting Documentation

Summary of the Contribution

We present Vortex a new family of one way hash functions that can produce message digests of 224, 256, 384 and 512 bits. The main idea behind the design of these hash functions is that we use well known algorithms that can support very fast diffusion in a small number of steps. We also balance the cryptographic strength that comes from iterating block cipher rounds with SBox substitution and diffusion (like Whirlpool) against the need to have a lightweight implementation with as small number of rounds as possible. We use a variable number of Rijndael rounds with a stronger key schedule. Our goal is not to protect a secret symmetric key but to support perfect mixing of the bits of the input into the hash value. Rijndael rounds are followed by our variant of Galois Field multiplication. This achieves cross-mixing between 128-bit or 256-bit sets. Our hash function uses the Enveloped Merkle-Damgård construction to support properties such as collision resistance, first and second pre-image resistance, pseudorandom oracle preservation and pseudorandom function preservation. We provide analytical results that demonstrate that the number of queries required for finding a collision with probability greater or equal to 0.5 in an ideal block cipher approximation of Vortex 256 is at least $1.18 \cdot 2^{122.55}$ if the attacker uses randomly selected message words. We also provide experimental results that indicate that the compression function of Vortex is not inferior to that of the SHA family regarding its capability to preserve the pseudorandom oracle property. We list a number of well known attacks and discuss how the Vortex design addresses them. The main strength of the Vortex design is that this hash function can demonstrate an expected performance of 2.2-2.5 cycles per byte in future processors with instruction set support for Rijndael rounds and carry-less multiplication. We provide arguments why we believe this is a trend in the industry. We also discuss how optimized assembly code can be written that demonstrates such performance.

1. Introduction

Guaranteeing message and code integrity is very important for the security of applications, operating systems and the network infrastructure of the future Internet. Protection against intentional alteration of data is typically supported using one way hash functions. A one way hash function is a mathematical construct that accepts as input a message of some length and returns a digest of much smaller length. One way hash functions are designed in such a way that it is computationally infeasible to find the input message by knowing only the digest. One way hash functions which have been in use today include algorithms like MD-5 and SHA1, SHA256, SHA384 and SHA512. The problem with using these algorithms is that they are time consuming when implemented in software. One way hash functions typically involve multiple shifts, XOR and ADD operations which they combine in multiple rounds in order to produce message digests. Because of this reason, one way hash functions consume a substantial number of processor clocks when executing, which limits their applicability to high speed secure network applications (e.g., 10 Gbps e-commerce transactions), or protection against malware (e.g., virus detection or hashed code execution).

In this submission document we describe Vortex: a novel hash family based on an alternative design approach. In our approach a family of one way hash functions is built from other well known security algorithms used as building blocks, as opposed to more primitive shift, rotate or XOR operations. The algorithms we choose in our design help with achieving fast mixing across a large number of input bits. Using the Merkle-Damgård [8, 15] and the Enveloped Merkle-Damgård [3] constructions as frameworks we construct a compression function from Rijndael rounds [1] and a novel merging technique based on Galois Field (GF(2)) multiplication. Using three or more successive Rijndael rounds we provide mixing across 128 bits. Using a merging function based on Galois Field (GF(2)) multiplication we provide mixing across sets of 128 bits. Perfect mixing is accomplished through combinations of Rijndael rounds and our merging function.

We provide analytical results that demonstrate that the number of queries required for finding a collision with probability greater or equal to 0.5 in an ideal block cipher approximation of Vortex 256 is at least $1.18 \cdot 2^{122.5}$ if the attacker uses randomly selected message words. We also provide experimental results that indicate that the compression function of Vortex is not inferior to that of the SHA family regarding its capability to preserve the pseudorandom oracle property. We have conducted experiments calculating statistical properties such as the collision probability, hamming weight, distribution function, and correlation matrix of outputs coming from randomly selected messages as well as single bit differentials superimposed on random messages. Our results indicate that there is no experimental evidence that Vortex is inferior when compared to the SHA-2 family in terms of its security. Performance-wise, however, the difference can be substantial. For example our SHA 256 implementation operates at 21 cycles per byte on an Intel® Core 2 Duo processor. The Vortex family is expected to operate at a speed between 2.2-2.5 cycles per byte in future processors with instruction set support for Rijndael round computation and Galois Field (GF(2)) multiplication. We provide

arguments why we believe this is a trend in the industry. We also discuss how optimized assembly code can be written that demonstrates such performance. We conclude by listing a number of well known attacks and how the Vortex design addresses them.

The document is structured as follows: In Section 2 we describe the design methodology of the Vortex family. In Section 3 we describe the algorithm. In Section 4, we provide qualitative arguments as well as analytical and experimental evidence for the security of Vortex. In Section 5 we present our performance analysis. In Section 6 we discuss related work. Finally in Section 7 we provide concluding remarks. The constant generation algorithm of Vortex is given in Appendix A and a list of intermediate values for a known answer test vector is given at the Appendix B. The rather lengthy proof of Lemma 1 used for the derivation of the main security proof is given in Appendix C.

2. Design Methodology

Vortex represents a new family of one way hash functions that can produce message digests of 224, 256, 384 and 512 bits. The main idea behind the design of these hash functions is that we use known algorithms that can support very fast diffusion in a small number of steps. Our intent is to allow each bit of an input block to affect all bits of a hash after a small number of computations.

The algorithms we use in our design are:

- The Rijndael round due to its capability to perform very fast mixing across 32-bits as a stand-alone operation and 128 bits or 256 bits if combined with at least one more round; and:
- A variant of Galois Field (GF(2)) multiplication due to its capability to cross mix bits of different sets (i.e., the input operands) in a manner that is cryptographically stronger than other simpler schemes (e.g., Feistel reordering proposed in modes like MDC-2 [23]).

We also balance the cryptographic strength that comes from iterating block cipher rounds with SBox substitution and diffusion (like Whirlpool) against the need to have a lightweight implementation with as small number of rounds as possible. We use a variable number of Rijndael rounds with stronger key schedule. The number of rounds is a tunable parameter. The best tradeoff between security and performance for the Vortex family comes when the number of rounds is greater or equal to 3. The design threshold of 3 comes from the fact that 2 rounds is the bare minimum number needed for 128-bit wide mixing. One more round is considered as a safety margin. The authors are aware that fewer than 10 Rijndael round transformations can be distinguished from random permutations in several ways. For this reason our design introduces a new key schedule algorithm for compensating for the security lost from reducing the number of Rijndael rounds in Vortex. In any case our design is open for introducing more rounds if proven necessary in the future. Rijndael rounds are followed by our variant of Galois Field

multiplication. This achieves cross-mixing between 128-bit or 256-bit sets. Our transformation is not simple carry-less multiplication but combines bit reordering operations, XORs and additions with carries. In this way our variant of Galois Field multiplication:

- achieves better diffusion than the straightforward carry-less multiplication between the 128-bit or 256-bit inputs
- is a non-commutative operation protecting against attacks based on swapping the order of the chaining variables in the processing of a message.

Our family of one way hash functions uses the Rijndael round as specified in [7]. Vortex 224 and Vortex 256 use Rijndael 128 rounds. Vortex384 and Vortex 512 use Rijndael 256 rounds. Rijndael 128 round is the round algorithm of the Advanced Encryption Standard (AES) as specified in the standard FIPS-197.

3. Algorithm Description

Endianness and Notation

Unless stated explicitly, the Vortex algorithm specification is independent of the endianness of the machine where Vortex is implemented. In the specification that follows we use the term 'least significant' to refer to the unit of information (whether bit, byte or word) with the smallest index. We also use the term 'most significant' to refer to the unit of information with the greatest index. For example in the bit representation of a 128-bit variable $A = [a_{127}a_{126}...a_0]$, a_0 is the least significant bit whereas a_{127} is the most significant bit.

Mathematical Operations

Mathematical operations used by the Vortex specification are listed below. These operations are: (i) bit-wise exclusive OR (XOR) (ii) addition modulo-2^{64}; and (iii) carry-less multiplication (iv) substitution box *SBox*() (v) Rijndael round *R*()

Bit-wise XOR is denoted by '$\oplus$' and defined as follows: Let the inputs to the operation are *A* and *B* consisting of *N* bits each:

$$A = [a_{N-1}a_{N-2}...a_0], \quad B = [b_{N-1}b_{N-2}...b_0] \tag{1}$$

The result of the operation $C = A \oplus B = [c_{N-1}c_{N-2}...c_0]$ is an *N*-bit number defined as:

$$c_i = \begin{cases} 0, & if \ \ a_i = b_i \\ 1, & otherwise \end{cases}, \quad 0 \le i \le N-1 \tag{2}$$

Addition modulo-2^{64} is denoted by '$\boxplus$' and defined as follows: Let the inputs to the operation are A and B consisting of M 64-bit words each:

$$A = [A_{M-1} : A_{M-2} : ... : A_0], \quad B = [B_{M-1} : B_{M-2} : ... : B_0] \tag{3}$$

The result of the operation $C = A \boxplus B = [C_{M-1} : C_{M-2} : ... : C_0]$ is an M-word number defined as:

$$C_i = (A_i + B_i) \mod 2^{64}, \quad 0 \le i \le M-1 \tag{4}$$

Carry-less multiplication is denoted by '$\otimes$' and defined as follows: Let the inputs to the operation are A and B consisting of N bits each:

$$A = [a_{N-1} a_{N-2} ... a_0], \quad B = [b_{N-1} b_{N-2} ... b_0] \tag{5}$$

The result of the operation $C = [c_{2N-2} c_{2N-2} ... c_0]$ is an $2N$-1 bit number. The bits of the output C result from the following logic functions of the bits of the inputs A and B:

$$c_i = \bigoplus_{j=0}^{i} a_j b_{i-j}, \quad 0 \le i \le N-1 \tag{6}$$

and:

$$c_i = \bigoplus_{j=i-N+1}^{N-1} a_j b_{i-j}, \quad N \le i \le 2N-2 \tag{7}$$

The SBox() transformation is defined as in the symmetric encryption standard FIPS-197 (AES). Let the input operand be A, defined a sequence of M bytes:

$$A = [A_{M-1} : A_{M-2} : ... : A_0] \tag{8}$$

The result of the transformation on A, is C = SBox(A) = $[C_{M-1} : C_{M-2} : ... : C_0]$ where

$$C_i = \text{sbox}(A_i) = A_T(M_I(A_i)), \quad 0 \le i < M-1 \tag{9}$$

The transformation sbox() is the AES substitution box applied on a single byte. Such transformation consists of two stages $M_I(a)$ and $A_T(a)$. In the first stage $M_I(a)$, each byte a is replaced by its multiplicative inverse in the finite field GF(2^8). This finite field is defined by the irreducible polynomial 0x11B (or 100011011b in binary notation). Additions and multiplications in GF(2^8) are carry-less and results are represented modulo

0x11B (or 100011011b). The second stage $A_T(a)$ replaces the value of each byte a with another byte value according to a bit-linear transform plus a constant. Specifically, every bit a_i of byte a is substituted by another bit a_i^+ according to the following formula:

$$a_i^+ = a_i \oplus a_{(i+4) \bmod 8} \oplus a_{(i+5) \bmod 8} \oplus a_{(i+6) \bmod 8} \oplus a_{(i+7) \bmod 8} \oplus w_i, \ 0 \le i \le 7 \tag{10}$$

where by '$\oplus$' we mean the XOR logical operation, and w_i is the i^{th} bit of the value 0x63.

The Rijndael round transformation $R()$ applies to inputs A and B consisting of M bytes each:

$$A = [A_{M-1} : A_{M-2} : \ldots : A_0], \quad B = [B_{M-1} : B_{M-2} : \ldots : B_0] \tag{11}$$

where $M = 16$ (Rijndael 128) or $M = 32$ (Rijndael 256). Input A denotes the round state whereas input B denotes the round key. The result of the transformation $C = R(A, B)$ is defined as:

$$C = M_C(\text{SBox}(S_R(A))) \oplus B \tag{12}$$

where the transformation $S_R(A)$ called 'Shift Rows' is a byte permutation on A. $S_R(A)$ is defined as:

$$S_R(A_i) = A_{(i+(i \bmod 4)\cdot 4) \bmod M} \tag{13}$$

The transformation $M_C(A)$ called 'Mix Columns' modifies the values of sequences of 4 adjacent bytes from A of the form $[A_{4\cdot i} : A_{4\cdot i+1} : A_{4\cdot i+2} : A_{4\cdot i+3}]$, $0 \le i < 4$, using matrix multiplication in the finite field GF(2^8).

$$\begin{bmatrix} M_C(A_{4\cdot i}) \\ M_C(A_{4\cdot i+1}) \\ M_C(A_{4\cdot i+2}) \\ M_C(A_{4\cdot i+3}) \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} A_{4\cdot i} \\ A_{4\cdot i+1} \\ A_{4\cdot i+2} \\ A_{4\cdot i+3} \end{bmatrix} \tag{14}$$

where the irreducible polynomial defining GF(2^8) is the same as the one used in the SBox() transformation (0x11B or 100011011b).

Block Length and Padding

Vortex processes an input stream as a sequence of blocks. A block is defined as a sequence of bits of specific length. The length of all blocks but the last is N bits. The length of the last block is $N/2$ bits. For Vortex 224 and Vortex 256 N=512. For Vortex 384 and Vortex 512 N=1024. The stream is padded with a bit value equal to'1' following the most significant bit of the input stream. The stream may be further padded with bits

of value equal to '0' so that its length becomes equal to (k+0.5) $\cdot N$ - N/8 for some non-negative integer k. If the length of the stream is already equal to (k+0.5) $\cdot N$ - N/8 for some k, the stream is not padded with zeros. Finally, the stream is padded with N/8 bits indicating the length of the stream. The length of the stream is expressed in bits and denoted using the little endian format. This means that the bit following the previous padding stages is the least significant bit of the value of the stream length.

Tunable Parameters

The tunable parameters of the Vortex algorithm include: (i) the number of rounds N_R used by an internal block cipher based on the Rijndael round R() transformation (see below); (ii) the degree of the diffusion D_F which affects the number of times each bit of the input stream is diffused over all bits of the output digest; and (iii) the type of multiplication algorithm M_T employed the Vortex merging function $V_M^{(A)}(a)$ described below. If $M_T = 0$ then multiplication is carry-less, else if $M_T = 1$, multiplication is integer.

Other tunable parameters include an initial value of the chaining variable $A_0 \,||\, B_0$ of the algorithm and a final tweak value $T_A \,||\, T_B$ both of which are explained below. By '||' we mean concatenation. $A_0 \,||\, B_0$ and $T_A \,||\, T_B$ are of size N/2 bits. The user of the Vortex algorithm can either set $A_0 \,||\, B_0$ and $T_A \,||\, T_B$ to some constants like the ones specified in the Appendix or to some randomly generated values. In this case Vortex operates as a pseudorandom function family, where for a different random pair of $A_0 \,||\, B_0$ and $T_A \,||\, T_B$ one obtains a different cryptographic hash function (which should ideally not be distinguishable from a random function). What is important in the specification of $A_0 \,||\, B_0$ and $T_A \,||\, T_B$ is that the chaining variable value $A_0 \,||\, B_0$ should always be different from the final tweak $T_A \,||\, T_B$.

Domain Extension Transform

Vortex operates on a chaining variable resulting from the concatenation of two N/4-bit variables A and B initialized to $A_0 \,||\, B_0$. Vortex also uses a tweak value consisting of the concatenation of variables T_A and T_B. T_A and T_B are N/4 bits long. To support collision resistance as well as pseudorandom function and pseudorandom oracle preservation, Vortex uses the Enveloped Merkle-Damgård construction as its domain extension transform. The Enveloped Merkle-Damgård construction is shown in Figure 1.

Each padded input stream consists of k blocks of size N bits for some non-negative integer k and a last block of size N/2 bits. Each block B_i (except for the last) consists of 4 words of size N/4 bits: $B_i = [W_{4 \cdot i+3} : W_{4 \cdot i+2} : W_{4 \cdot i+1} : W_{4 \cdot i}]$, $0 \le i \le k$-1. The last block B_k consists of 2 words: $B_k = [W_{4 \cdot k+1} : W_{4 \cdot k}]$. The compression function used by the Enveloped Merkle-Damgård construction, called 'Vortex block' works as follows: It

accepts as input the previous value of a chaining variable $A_i \| B_i$ and an input block B_i, $0 \le i \le k$-1. It returns an updated value of the chaining variable $A_{i+1} \| B_{i+1}$.

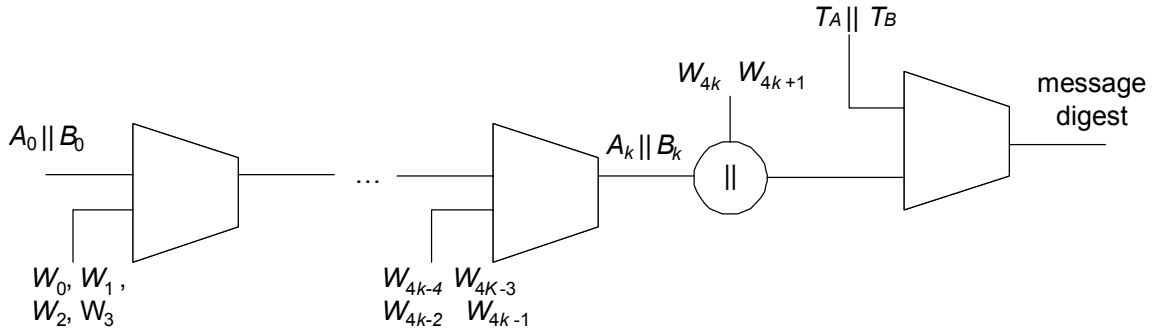

Figure 1: Vortex as an Enveloped Merkle-Damgård construction

The processing done on the last block differs from the processing done on other blocks. For the last block, the compression function uses the tweak value $T_A \| T_B$ as a chaining variable and the concatenation of $[A_k : B_k]$ and $[W_{4\cdot k+1} : W_{4\cdot k}]$ as input block. For Vortex 256 and Vortex 512, the message digest resulting from the input stream is equal to the final value of the chaining variable $[A_{k+1} : B_{k+1}]$. For Vortex 224 and Vortex 384, the message digest resulting from the input stream is equal to the 224 and 284 least significant bits of $[A_{k+1} : B_{k+1}]$ respectively.

Vortex Block

The Vortex block algorithm incorporates two repetitions of an algorithm called 'Vortex - sub-block'. Such structure is shown in Figure 2. The first repetition of Vortex sub-block accepts as input the chaining variable $A_i \| B_i$ and two least significant input block words $W_{4\cdot i}$, $W_{4\cdot i+1}$. It returns an intermediate value for the chaining variable $A \| B$. The second repetition of Vortex sub-block accepts as input the intermediate value of the chaining variable $A \| B$ and two most significant input block words $W_{4\cdot i+2}$, $W_{4\cdot i+3}$. It returns an update on the chaining variable $A_{i+1} \| B_{i+1}$. The Vortex block algorithm is:

Vortex block($A_i, B_i, W_{4\cdot i}, W_{4\cdot i+1}, W_{4\cdot i+2}, W_{4\cdot i+3}$)
begin
$A \| B \leftarrow$ **Vortex sub-block**($A_i, B_i, W_{4\cdot i}, W_{4\cdot i+1}$) // uses $W_{4\cdot i}, W_{4\cdot i+1}$
$A_{i+1} \| B_{i+1} \leftarrow$ **Vortex sub-block**($A, B, W_{4\cdot i+2}, W_{4\cdot i+3}$) // uses $W_{4\cdot i+2}, W_{4\cdot i+3}$
return $A_{i+1} \| B_{i+1}$
End

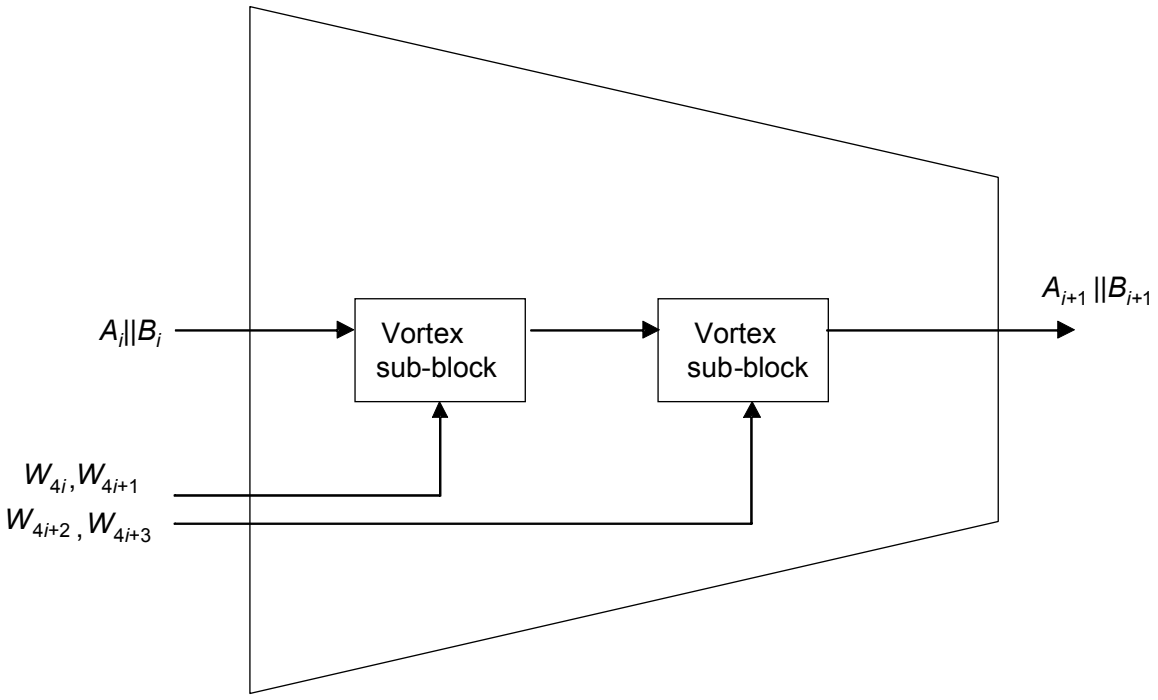

Figure 2: Structure of Vortex block

Vortex Sub-block

With the exception of the last sub-block (see below), the algorithm for processing a Vortex sub-block is the following:

Vortex sub-block(A, B, W_0, W_1)
begin

; W_0 is the first word of the current sub-block to be processed

$A \leftarrow \widetilde{A}_A(W_0) \oplus W_0$

$B \leftarrow \widetilde{A}_B(W_0) \oplus W_0$

$A \| B \leftarrow V_M^{(A)}(A, B)$

;W_1 is the second word of the current sub-block to be processed

$A \leftarrow \widetilde{A}_A(W_1) \oplus W_1$

$B \leftarrow \widetilde{A}_B(W_1) \oplus W_1$

$A \| B \leftarrow V_M^{(A)}(A, B)$

return $A \,||\, B$

end

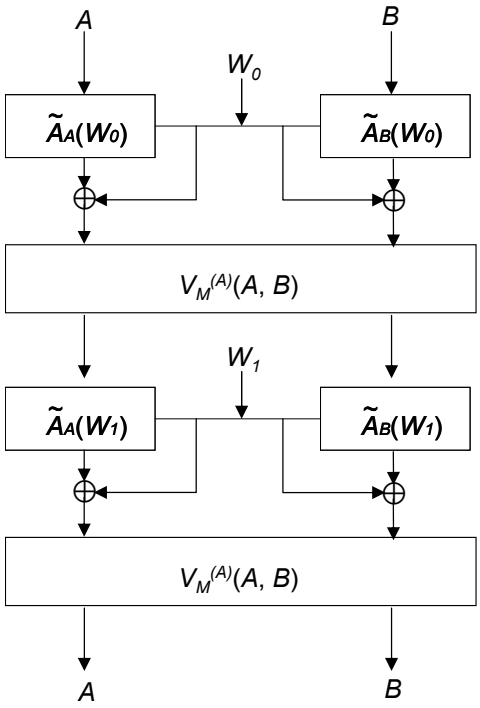

Figure 3: Vortex sub-block

The structure of the Vortex sub-block is shown in Figure 3. The Vortex sub-block is built upon two mathematical functions: The transformation $\tilde{A}_K(x)$ called 'A-Rijndael', which is a block cipher based on Rijndael rounds, and the merging function $V_M^{(A)}(A, B)$. There are four instances of the transformation $\tilde{A}_K(x)$ in the Vortex sub-block. Each instance is wrapped using a Matyas-Meyer-Oseas structure to make the transformation non-reversible. The first two instances process input word W_0. The other two instances process the input word W_1. W_0 is the least significant word of the current sub-block to be processed. Instances of $\tilde{A}_K(x)$ that accept the same input word processes a different variable from among A, B. Each instance treats its input variable A or B as a key and its input word, which is one from W_0 or W_1 as plaintext, as it is the norm in the a Matyas-Meyer-Oseas structure. The merging function $V_M^{(A)}(A, B)$ combines the outputs of the two instances of $V_M^{(A)}(A, B)$ into a new value of $A||B$.

A-Rijndael Transformation

The A-Rijndael transformation $\tilde{A}_K(x)$ is a high performance block cipher based on Rijndael rounds that encrypts x, which is $N/4$ bits long, using the key K which is also $N/4$ bits long. $\tilde{A}_K(x)$ uses a tunable number of Rijndael rounds which we symbolize as N_R. For $N/4$=128 rounds are as specified in AES, FIPS-197 [1]. Each Rijndael round $R()$ consists of an SBox() substitution phase, a 'Shift Rows' transformation, a 'Mix Columns'

transformation and a round key addition in GF(2). The key schedule algorithm used by $\widetilde{A}_K(x)$ is different from that of Rijndael. $\widetilde{A}_K(x)$ uses a variable number N_R of N/4-bit wide *Rcon* values RC_1, RC_2 … RC_{NR} to derive N_R round keys RK_1, RK_2 … RK_{NR} as follows:

$RK_1 \leftarrow \mathsf{Perm}(\text{SBox}(K \boxplus RC_1))$
$RK_2 \leftarrow \mathsf{Perm}(\text{SBox}(RK_1 \boxplus RC_2))$
...
$RK_{NR} \leftarrow \mathsf{Perm}(\text{SBox}(RK_{NR\text{-}1} \boxplus RC_{NR}))$

where Perm() is a byte permutation and by '⊞' we mean addition modulo 2^{64}. In this specification and prototype implementation Perm() is equal to the identity function. Other byte permutations can be considered though if deemed necessary. The SBox() transformation in the key schedule is applied on N/32 bytes, i.e., N/4 bits (i.e., 128 bits or 16 bytes for Vortex 256 and 256 bits or 32 bytes for Vortex 512). A single Rijndael round performs diffusion across 32 bits. This is accomplished through the combination of the SBox() and Mix Columns transformations. Two Rijndael rounds diffuse across 128 bits. This is accomplished through the combination of the subsequent Shift Rows and Mix Columns transformations. Three or more rounds further strengthen the diffusion performed. The *Rcon* values are set to constant values. The algorithm for generating these constants is described in Appendix A.

Merging Function $V_M^{(A)}(A,B)$

The merging function $V_M^{(A)}(A,B)$ is shown in Figure 4. If the multiplication type M_T is 0 (carry-less multiplication) $V_M^{(A)}(A,B)$ operates as follows:

$V_M^{(A)}(A,B)$
begin
    let $A = [A_1 : A_0]$ // A_1, A_0 are N/8 bit words
    let $B = [B_1 : B_0]$ // B_1, B_0 are N/8 bit words
    $O \leftarrow A_0 \otimes B_1$
    $I \leftarrow A_1 \otimes B_0$
    let $I = [I_1, I_0]$ // I_1, I_0 are N/8 bit words
    let $O = [O_1, O_0]$ // O_1, O_0 are N/8 bit words
    return $[B_1 \boxplus I_1 : B_0 \boxplus O_0 : A_1 \oplus O_1 : A_0 \oplus I_0]$
end

where by '⊞' we addition modulo 2^{64}, and '⊗' we mean carry-less multiplication.

If the multiplication type M_T is 1 (integer multiplication) $V_M^{(A)}(A,B)$ operates as follows:

$V_M^{(A)}(A, B)$

begin

let $A = [A_1 : A_0]$ // A_1, A_0 are $N/8$ bit words

let $B = [B_1 : B_0]$ // B_1, B_0 are $N/8$ bit words

$O \leftarrow A_0 \cdot B_1$

$I \leftarrow A_1 \cdot B_0$

let $I = [I_1, I_0]$ // I_1, I_0 are $N/8$ bit words

let $O = [O_1, O_0]$ // O_1, O_0 are $N/8$ bit words

return $[B_1 \boxplus I_1 : B_0 \boxplus O_0 : A_1 \oplus O_1 : A_0 \oplus I_0]$

end

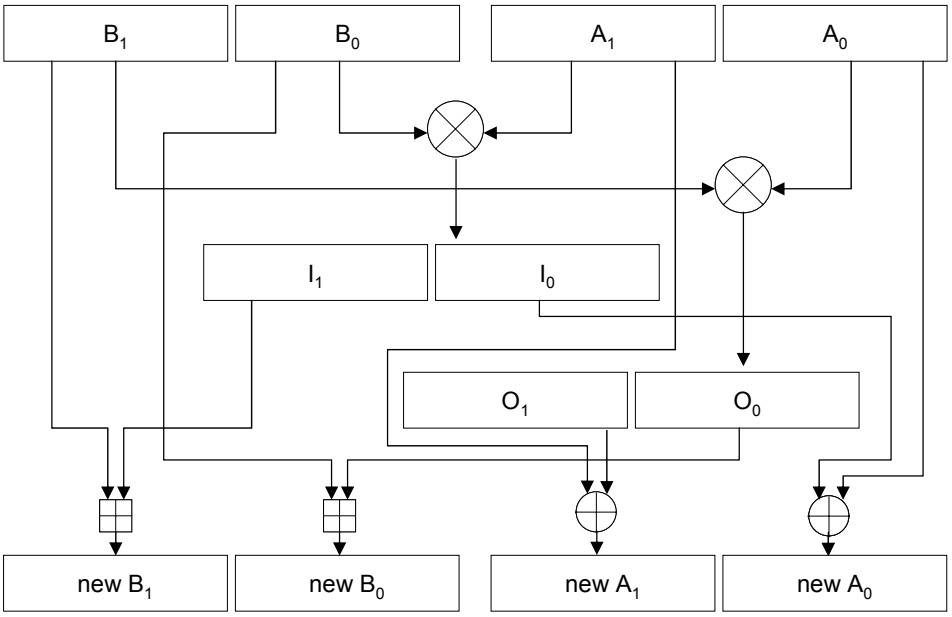

Figure 4: The Merging Function of Vortex (M_T=0)

The merging function is based on multiplication (carry-less or integer). Our merging function makes sure that the bits of A impact the bits of B and vice versa. In fact, each bit of one variable affects a significant number of the bits of the other variable in a non-linear manner. This makes our design better than a straightforward XOR or other simple mathematical operation.

Setting M_t = 0 (carry-less multiplication) is the default configuration of Vortex. The reason why Vortex uses carry-less multiplication by default is because it is easier to assert analytically about the collision resistance and pre-image resistance of the hash as explained in the next section. On the other hand using an integer multiplier in the merging function increases the performance of the hash (not all processor architectures

have a carry-less multiplier), increases the non-linearity of merging, but makes the security of the scheme more difficult to prove.

Last Vortex Sub-block

One can also observe that even though our merging function is strong cryptographically due to the mixing it provides, it does not accomplish perfect mixing by itself. This is because each bit of A or B affects a large number of bits of the other variable but not all of them. Perfect mixing is accomplished by the Rijndael rounds that follow our merging function. So, for a pair of input words W_0, W_1 perfect mixing is accomplished after a sequence of Rijndael rounds (mix across $N/4$ bits), merging using Galois Field multiplication (cross-mix across $N/4$ bit sets but not perfect mixing) and another set of Rijndael rounds as part of the sub-block processing to follow.

The total number of times every bit is diffused over all bits of the hash is determined by the number of sequences of Rijndael rounds and merging found in the last Vortex sub-block; This is another tunable parameter of the hash called 'degree of diffusion' D_F. The algorithm for the last Vortex sub-block is given below:

Last Vortex sub-block(A, B, W_0, W_1, D_F)
begin

; W_0 is the first word of the last sub-block to be processed

$A \leftarrow \widetilde{A}_A(W_0) \oplus W_0$

$B \leftarrow \widetilde{A}_B(W_0) \oplus W_0$

$A \| B \leftarrow V_M^{(A)}(A, B)$

for $i \leftarrow 1$ to D_F **do**

; D_F is the degree of diffusion

;W_1 is the second word of the current sub-block to be processed

$A \leftarrow \widetilde{A}_A(W_1) \oplus W_1$

$B \leftarrow \widetilde{A}_B(W_1) \oplus W_1$

$A \| B \leftarrow V_M^{(A)}(A, B)$

return $A||B$

}

4. Security Analysis

The security of the Vortex family was investigated both analytically and experimentally by computing or measuring the collision probability, hamming weight, mean value, distribution and other statistical properties of outputs from random messages as well as single bit output differentials. For these experiments we compared the numbers we got from Vortex with numbers we got from the SHA family. No collision occurred in our

experiments. Our initial results indicate that there is no experimental evidence that Vortex is inferior in terms of its security properties when compared to the SHA family.

Qualitative Analysis

We argue that the Vortex family is at least as secure as the SHA family even though it uses smaller number of processing steps. There are several reasons for this. First Rijndael round is a good mixing function. The key used results from the current value of the chaining variable and hence is in most cases data dependent. Hence our scheme does not suffer from known attacks on compression functions that use a small set of keys [5]. The key schedule transformation of Vortex is stronger than Rijndael due to the fact that the SBox() transformation is applied across each 128- or 256-bit round key as opposed to 32 bits only and that round constants are added using integer addition modulo 2^{64} as opposed to XOR. It is the combination of two independent sources of non-linearities in the key schedule, i.e., addition with carries and inversion in GF(2^8) that strengthen the mixing performed by Vortex. We have investigated experimentally whether the security obtained by strengthening the key schedule of A-Rijndael can compensate for reducing its number of rounds (to increase performance). Our initial results indicate that, even with a 3 round A-Rijndael transformation, Vortex outputs message digests with satisfactory statistical properties.

Vortex uses a Matyas-Meyer-Oseas transformation, where they key is obtained from the chaining variable and not the message. Because of this reason Vortex is more secure against related key attacks. This is because the attacker can be in control of the message supplied as input to A-Rijndael but not the key. The merging function of Vortex combines linear (XORs) and non-linear (adds with carries) transformations with 64-bit or 128-bit multiplication building blocks. This operation is non-commutative and when combined with previous and subsequent Rijndael rounds and Galois Field multiplication achieves perfect mixing across $N/2$ bits. By designing the merging function to be non-commutative we destroy any symmetry in the computation of the Vortex sub-block that could be a potential source of collision. If Vortex was designed such that its merging function is commutative, then an attacker could create a collision by generating a message that swaps the position of chaining variables A and B as compared to another given message.

A more thorough analytical study on the security of the Vortex family is described in the next section. Specifically, we show that the number of queries required for finding a collision with probability greater or equal to 0.5 in an ideal block cipher approximation of A-Rijndael is at least $1.18 \cdot 2^{122.5}$. A part of this work we developed a methodology for computing the collision resistance and the first pre-image resistance of our construction based on the divide-and-conquer approach that was first used in the study of the MDC-2 mode by Steinberger [23]. Such approach helps with reasoning about the collision and pre-image resistance of specific components of hash functions. Components of hash functions include adders, shifters, XORs, S-Boxes, linear diffusers, bit permutations etc. Whereas our merging function is more complex than the MDC-2 mode of operation it can be analyzed due to the fact that it combines relatively simple building blocks (i.e.,

multipliers adders and XORs). In addition when M_T = 0 multipliers are carry-less accepting small size input operands (i.e., 64 or 128 bits). These facts make the collision and pre-image resistance of our construction easier to compute than MDC-2.

The current design choices have been made to balance the security of Vortex with performance. There are several possible extensions that can be made to the Vortex design though. As part of future work, we need to determine whether the presence of simple carry-less or integer multiplication is sufficient in the merging function or not. Any non-zero operand multiplied with zero results in zero. Such fact marginally increases the collision probability associated with our merging function as explained below (we obtain 122 bits of collision resistance as opposed to128). If this is proven to be a design deficiency, it can be potentially corrected with simple modifications to the algorithm. For example, a single multiplication can be replaced by two multiplications. In each of the two multiplications, operands are XOR-ed with a correcting constant and the results of the multiplications are merged with each other.

A careful observer can also see that when the multiplication type is 0 (carry-less) the most significant bit of A is not affected by the merging function whereas the most significant bit of B is only changed by the carry value, and so it remains the same with probability 0.5. Diffusion in the most significant bit position of A is completed by subsequent stages of Rijndael rounds and merging. This is one of the reasons why the last Vortex sub-block executes several times.

Theoretical Analysis

In what follows we provide an analytical argument for the collision and pre-image resistance of the Vortex algorithm. For our analysis we replace the A-Rijndael transformation with an ideal block cipher. An ideal block cipher is computationally indistinguishable from a random permutation given a secret randomly chosen key. The rationale behind such replacement is that the number of rounds N_R of A-Rijndael can be se to any appropriate value since it is a tunable parameter of the algorithm. Once the number of rounds is sufficiently large, A-Rijndael does approximate an ideal block cipher, hence we can safely do the replacement. We also restrict our analysis for merging functions with carry-less multiplication (M_T = 0). The reason why is because it is much easier to analyze the behavior of a carry-less multiplier as compared to an integer due to the absence of carry propagation.

In what follows we define a ‘query’ as a sequence of A-Rijndael transformations followed by merging. A query is part of the Vortex sub-block algorithm.

Query(A, B, W_0)
begin
; W_0 is the word of the current sub-block to be processed

$A \leftarrow \widetilde{A}_A(W_0) \oplus W_0$

$B \leftarrow \widetilde{A}_B(W_0) \oplus W_0$

$A \| B \leftarrow V_M^{(A)}(A, B)$

return $A \| B$

end

Theorem 1: The number of queries required for finding a collision with probability greater or equal to 0.5 in an ideal block cipher approximation of A-Rijndael is at least $1.18 \cdot 2^{122.5}$ for Vortex 224 and Vortex 256, if the attacker uses randomly chosen message words for the queries.

To prove the theorem above we investigate the behavior of the 64-bit carry-less multiplier which is the main mixing component of the Vortex merging function. We show that if the input to the carry-less multiplier is uniformly distributed then the output is almost uniformly distributed too. Let the inputs to a 64-bit carry-less multiplier by X and Y:

$$X = [x_{63} x_{62} ... x_0], \quad Y = [y_{63} y_{62} ... y_0] \tag{15}$$

Let's also assume that X and Y are uniformly distributed:

$$\Pr(X = \widetilde{X}) = \Pr(Y = \widetilde{Y}) = 2^{-64}, \ \ \forall \widetilde{X}, \widetilde{Y} \in [0, \ 2^{64} - 1] \tag{16}$$

We denote the output of carry-less multiplication as [W:Z]:

$$[W : Z] = X \otimes Y \tag{17}$$

where W is a 63 bit word and Z is a 64-bit word:

$$W = [w_{62} w_{61} ... w_0], \quad Z = [z_{63} z_{62} ... z_0] \tag{18}$$

In what follows we state a useful Lemma regarding the probability distribution of $[W : Z]$

Lemma 1: Let $[W : Z] = X \otimes Y$ be the result of the carry-less multiplication of quantities X and Y defined as in Eq. (15) and distributed as in Eq. (16). Then the probability that $[W : Z]$ takes a specific value $[\widetilde{W} : \widetilde{Z}]$ in the set [0, 2^{128}-1] is bounded by:

$$\Pr([W : Z] = [\widetilde{W} : \widetilde{Z}]) \le 2^{-122.5}, \ \ \forall [\widetilde{W} : \widetilde{Z}] \in [0, \ 2^{128} - 1] \tag{19}$$

Moreover, the probability that Z takes a specific value $\widetilde{Z}$ in the set [0, 2^{64}-1] is bounded by:

$$\Pr(Z = \widetilde{Z}) \le 2^{-61.66}, \ \forall \widetilde{Z} \in [0, \ 2^{64} - 1] \tag{20}$$

and the probability that W takes a specific value $\widetilde{W}$ in the set [0, 2^{64}-1] is bounded by:

$$\Pr(W = \widetilde{W}) \le 2^{-60.83}, \ \forall \widetilde{W} \in [0, \ 2^{64} - 1] \tag{21}$$

Proof of Lemma 1: is provided in Appendix C.

To prove Theorem 1 we further show that the output of the 'Query' algorithm Query(A, B, W_0) is almost uniformly distributed if at least the input word W_0 is uniformly distributed.

Lemma 2: Let $[C : D]$= Query(A, B, W_0) be the output of the query algorithm on N/4 bit quantities A, B, W_0. Let N=512. Let also W_0 be uniformly distributed and the A-Rijndael transformation used by Query() replaced by an ideal block cipher. Then the probability that $[C : D]$ takes a specific value $[\widetilde{C} : \widetilde{D}]$ in the set [0, 2^{256}-1] is bounded by:

$$\Pr([C : D] = [\widetilde{C} : \widetilde{D}]) \le 2^{-245}, \ \forall [\widetilde{C} : \widetilde{D}] \in [0, \ 2^{256} - 1] \tag{22}$$

Proof follows from Lemma 1. The behavior of the Query algorithm is illustrated in Figure 5.

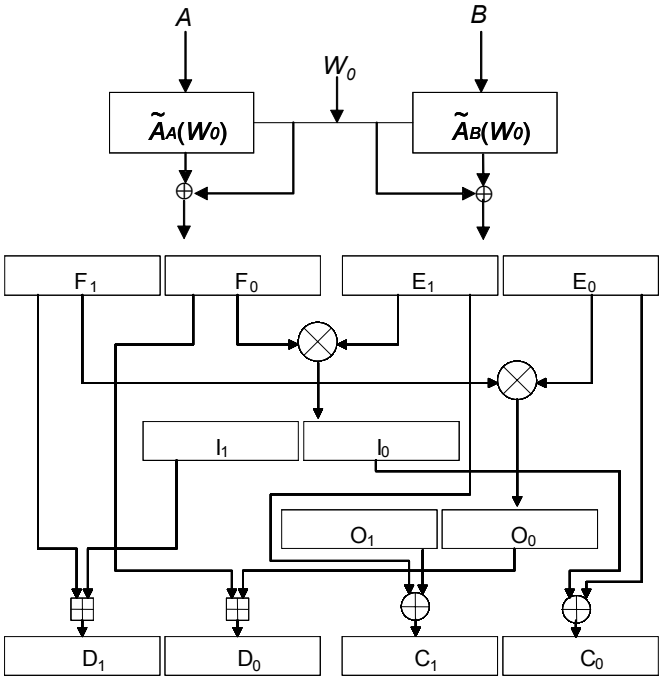

Figure 5: Behavior of the Query Algorithm

Since the input word is uniformly distributed and the A-Rijndael transformation is replaced by an ideal block cipher, the outputs $E=[E_1 : E_0]$ and $F=[F_1 : F_0]$ of the Matyas-Meyer-Oseas structures of the Query algorithm are also uniformly distributed. Because of this reason the probability distributions of the inner and outer products $I=[I_1 : I_0]$ and $O=[O_1 : O_0]$ are bounded according to Lemma 1:

$$\Pr(I_0 = \widetilde{I}_0) \le 2^{-61.66}, \ \forall \widetilde{I}_0 \in [0,\ 2^{64}-1] \tag{23}$$

$$\Pr(I_1 = \widetilde{I}_1) \le 2^{-60.83}, \ \forall \widetilde{I}_1 \in [0,\ 2^{64}-1] \tag{24}$$

$$\Pr(O_0 = \widetilde{O}_0) \le 2^{-61.66}, \ \forall \widetilde{O}_0 \in [0,\ 2^{64}-1] \tag{25}$$

$$\Pr(O_1 = \widetilde{O}_1) \le 2^{-60.83}, \ \forall \widetilde{O}_1 \in [0,\ 2^{64}-1] \tag{26}$$

Hence Lemma 2 is proven as follows:

$$\begin{aligned}
&\Pr([C : D] = [\widetilde{C} : \widetilde{D}]) = \\
&\Pr(C_0 = \widetilde{C}_0) \cdot \Pr(C_1 = \widetilde{C}_1) \cdot \\
&\Pr(D_0 = \widetilde{D}_0) \cdot \Pr(C_1 = \widetilde{D}_1) = \\
&\Big(\sum_{\widetilde{E}_0, \widetilde{I}_0 \in [0, 2^{64}-1],\ \widetilde{E}_0 \oplus \widetilde{I}_0 = \widetilde{C}_0} \Pr(E_0 = \widetilde{E}_0) \cdot \Pr(I_0 = \widetilde{I}_0)\Big) \cdot \\
&\Big(\sum_{\widetilde{E}_1, \widetilde{O}_1 \in [0, 2^{64}-1],\ \widetilde{E}_1 \oplus \widetilde{O}_1 = \widetilde{C}_1} \Pr(E_1 = \widetilde{E}_1) \cdot \Pr(O_1 = \widetilde{O}_1)\Big) \cdot \\
&\Big(\sum_{\widetilde{F}_0, \widetilde{O}_0 \in [0, 2^{64}-1],\ \widetilde{F}_0 + \widetilde{O}_0 = \widetilde{D}_0 \pmod{2^{64}}} \Pr(F_0 = \widetilde{F}_0) \cdot \Pr(O_0 = \widetilde{O}_0)\Big) \cdot \\
&\Big(\sum_{\widetilde{F}_1, \widetilde{I}_1 \in [0, 2^{64}-1],\ \widetilde{F}_1 + \widetilde{I}_1 = \widetilde{D}_1 \pmod{2^{64}}} \Pr(F_1 = \widetilde{F}_1) \cdot \Pr(I_1 = \widetilde{I}_1)\Big) \le \\
&2^{64} \cdot \frac{1}{2^{64}} \cdot \frac{1}{2^{61.66}} \cdot 2^{64} \cdot \frac{1}{2^{64}} \cdot \frac{1}{2^{60.83}} \cdot \\
&2^{64} \cdot \frac{1}{2^{64}} \cdot \frac{1}{2^{61.66}} \cdot 2^{64} \cdot \frac{1}{2^{64}} \cdot \frac{1}{2^{60.83}} = \\
&2^{-245}
\end{aligned} \tag{27}$$

Lemma 2 indicates that the Vortex 256 algorithm offers at least 245 bits of first pre-image resistance. The reason why we do not get the ideal 256 bit security is because of the zero accumulation point of the multiplier units employed by the Vortex merging function. As it is shown in Appendix C (proof of lemma 1) the byte distribution probabilities for the output of the carry-less multiplier are all bounded by 2^{-8} apart from those associated with the most and least significant byte positions of the output. For these byte positions the bounds are $2^{-4.83}$ and $2^{-5.68}$ respectively. These higher bounds come from the probabilities that the output bytes are equal to zero. As a result the Vortex

design cannot achieve the ideal security of 256 bits. However, there are several ways by which the design can be extended if this is deemed necessary.

From Lemma 1 and Lemma 2 we can now prove theorem 1 and conclude this theoretical analysis of our algorithm. We assume the presence of an adversary who chooses randomly selected messages as inputs to the Query() algorithm. The values of the chaining variables A and B are not under the explicit control of the adversary. Lemma 2 states that the probability that the output of Query(A, B, W_0) is equal to a specific value $[\widetilde{C} : \widetilde{D}]$ is bounded by 2^{-245}. This bound can be written as $2^{139}/2^{384}$. This means that from all possible 2^{384} triplets (A, B, W_0) there exists a set of no more than 2^{139} resulting in the same specific output $[\widetilde{C} : \widetilde{D}]$. This also means that there are at least 2^{245} sets of triplets (A, B, W_0) of cardinality less than or equal to 2^{139} where the triples of the same set result in the same output.

The probability $\hat{p}$ that the adversary has not found a collision after q queries is equal to the probability that the adversary has used a triplet from a different set in each of the q queries the adversary has made. To compute a lower bound for the number of queries the adversary needs to make so that $1-\hat{p} \geq 0.5$ we proceed as in any typical birthday attack. Considering $2^{245} = N_T$, we have:

$$\begin{aligned} \hat{p} &= 1 \cdot \left(\frac{N_T - 1}{N_T}\right) \cdot \left(\frac{N_T - 2}{N_T}\right) \cdot \ldots \cdot \left(\frac{N_T - q + 1}{N_T}\right) \\ &= (1 - \frac{1}{N_T}) \cdot (1 - \frac{2}{N_T}) \cdot \ldots \cdot (1 - \frac{q-1}{N_T}) \\ &\approx e^{-\frac{1+2+\ldots+q-1}{N_T}} = e^{-\frac{q(q-1)}{N_T}} \end{aligned} \tag{28}$$

Setting $1-\hat{p} = ½$ we get that:

$$e^{-\frac{q(q-1)}{N_T}} = \frac{1}{2} \Leftrightarrow q^2 - q = 2 \cdot \ln 2 \cdot N_T \Leftrightarrow q^2 - q - 2 \cdot \ln 2 \cdot N_T = 0 \tag{29}$$

The above equation has two roots $q = (1 \pm \sqrt{1 + 8 \cdot \ln 2 \cdot N_T}) / 2$ of which the positive is equal to $q = (1 + \sqrt{1 + 8 \cdot \ln 2 \cdot N_T}) / 2$.

For $N_T >> 1$, we get that $q = 1.18 \cdot \sqrt{N_T} = 1.18 \cdot 2^{122.5}$. Hence Theorem 1 is proven.

Experimental Analysis

To evaluate our algorithm we conducted a number of experiments hashing random messages and computing the statistical properties of the resulting digests as well as single bit output differentials. In this section we present our experimental results. The collision

resistance of the Vortex family was investigated experimentally by conducting a large number of experiments (2^{20}) hashing the Vortex specification document with random perturbations superimposed on it. No collision occurred in our experiments.

Another set of experiments was conducted in order to demonstrate how random the outputs of the Vortex hash are. For this purpose we first computed the Hamming weight of message digests resulting from the short and long message Known Answer Test vectors (KATs) provided by NIST. Our Hamming weight analysis results are illustrated below. For each value shown in Tables 1-5, the notation used is 'average ± standard' deviation.

Experiment	N_R**=3,** D_F**=5**	N_R**=5,** D_F**=5**	N_R**=7,** D_F**=5**	N_R**=10,** D_F**=5**
Short Messages, Vortex 224	112.1±7.4	111.8±7.5	112.3±7.6	112.0±7.4
Long Messages, Vortex 224	112.3±7.4	111.3±7.4	112.2±8.0	111.9±7.5
Short Messages, Vortex 256	128.1±8.1	127.8±8.0	128.3±8.1	128.0±7.8
Long Messages, Vortex 256	128.6±7.9	127.2±7.8	128.2±8.4	128.0±7.9
Short Messages, Vortex 384	191.8±9.8	192.2±9.9	192.3±9.8	192.1±9.9
Long Messages, Vortex 384	191.7±9.7	191.6±10.1	192.1±10.1	192.5±10.4
Short Messages, Vortex 512	255.7±11.3	256.2±11.5	256.1±11.4	256.0±11.4
Long Messages, Vortex 512	255.5±11.4	255.6±11.7	255.8±11.5	256.0±11.8

Table 1: Hamming Weight Analysis for Carry-less Multiplication (M_T=0)

Experiment	N_R**=3,** D_F**=5**	N_R**=5,** D_F**=5**	N_R**=7,** D_F**=5**	N_R**=10,** D_F**=5**
Short Messages, Vortex 224	112.1±7.3	111.9±7.5	112.1±7.6	111.8±7.4
Long Messages, Vortex 224	112.5±7.4	111.9±7.7	112.6±7.3	111.9±7.9
Short Messages, Vortex 256	128.0±7.8	127.9±7.9	128.2±8.1	127.9±8.0
Long Messages, Vortex 256	128.3±7.9	128.0±8.4	128.5±7.7	127.9±8.3
Short Messages, Vortex 384	192.3±10.2	192.2±10.0	192.1±9.9	192.1±10.0
Long Messages, Vortex 384	191.4±10.2	191.8±9.4	192.7±10.1	191.1±9.9
Short Messages, Vortex 512	256.4±11.8	256.0±11.4	256.1±11.3	256.2±11.4
Long Messages, Vortex 512	255.1±12.4	255.7±11.3	256.4±11.6	255.0±11.0

Table 2: Hamming Weight Analysis for Integer Multiplication (M_T=1)

A second set of experiments was conducted in order to measure the statistical properties of single bit output differentials. We computed the statistical mean and correlation matrix of single bit output differentials resulting from 16K random messages of size 256 bits. For the derivation of the differentials we XOR-ed a single bit perturbation mask with the value "1" being set in bit positions 0 to 255 to each input message. Then, we XOR-ed the outputs.

Experiment	**N_R=3, D_F=5**	**N_R=5, D_F=5**
Vortex 256 (1 bit)	8192.37 ± 64.03	8191.93 ± 63.99
SHA 256 (1 bit)	8191.99 ± 63.96	8192.6 ± 63.98
Vortex 256 (16 bits)	53689.75 ± 238.16	53687.54± 242.44
SHA 256 (16 bits)	53689.86 ± 238.91	53693.23± 241.05
Experiment	**N_R=7, D_F=5**	**N_R=10, D_F=5**
Vortex 256 (1 bit)	8191.69 ± 64.08	8192.03 ± 63.98
SHA 256 (1 bit)	8191.75 ± 63.93	8191.99 ± 63.85
Vortex 256 (16 bits)	53688.50 ± 242.03	53685.16 ± 240.04
SHA 256 (16 bits)	53682.48 ± 246.06	53681.98 ± 242.25

Table 3: Single Bit Differential Analysis: Scaled Mean Values across Output Differentials for Carry-less Multiplication (M_T=0)

Experiment	**N_R=3, D_F=5**	**N_R=5, D_F=5**
Vortex 256 (1 bit)	8192.46 ± 63.87	8191.81 ± 63.96
SHA 256 (1 bit)	8192.32 ± 63.92	8192.16 ± 64.22
Vortex 256 (16 bits)	53690.16 ± 245.51	53680.92 ± 239.86
SHA 256 (16 bits)	53690.73 ± 243.23	53689.34 ± 241.73
Experiment	**N_R=7, D_F=5**	**N_R=10, D_F=5**
Vortex 256 (1 bit)	8191.67 ± 63.91	8192.33 ± 64.24
SHA 256 (1 bit)	8191.96 ± 63.94	8191.76 ± 63.86
Vortex 256 (16 bits)	53678.77 ± 244.83	53687.57 ± 242.94
SHA 256 (16 bits)	53681.87.± 242.01	53688.66 ± 243.16

Table 4: Single Bit Differential Analysis: Scaled Mean Values across Output Differentials for Integer Multiplication (M_T=1)

The statistical means and correlation properties shown in Tables 1-5 were derived from these computed output differentials. We considered that either a single bit or a group of 16 consecutive bits in an output differential is a random signal. Computations were done to derive the statistical properties of these random signals over specific sets of output differentials. Each differential in a set was associated with a perturbation in a different bit position from 0 to 255, over the same random message.

Tables 3 and 4 show the mean values across all random signals coming from output differentials for M_T=0 and M_T=1 respectively. Table 5 shows the sum of the elements of the correlation matrix computed for the random signals described above. The results from our experiments were multiplied with appropriate scaling constants so that the comparison between Vortex 256 and SHA 256 is meaningful.

Experiment	**N_R=3, D_F=5, M_T=0**	**N_R=5, D_F=5, M_T=0**
Vortex 256 (16 bits)	147.99 ± 55.27	148.71 ± 55.90
SHA 256 (16 bits)	152.92 ± 56.23	150.82 ± 56.11
Experiment	**N_R=7, D_F=5, M_T=0**	**N_R=10, D_F=5, M_T=0**
Vortex 256 (16 bits)	151.43 ± 56.55	162.26 ± 59.04
SHA 256 (16 bits)	149.55 ± 55.88	154.34 ± 56.71
Experiment	**N_R=3, D_F=5, M_T=1**	**N_R=5, D_F=5, M_T=1**
Vortex 256 (16 bits)	150.31 ± 56.55	149.14 ± 55.41
SHA 256 (16 bits)	150.45 ± 57.09	147.32 ± 55.15
Experiment	**N_R=7, D_F=5, M_T=1**	**N_R=10, D_F=5, M_T=1**
Vortex 256 (16 bits)	154.68 ± 56.42	153.66 ± 57.21
SHA 256 (16 bits)	151.31 ± 56.74	150.29 ± 57.98

Table 5: Single Bit Differential Analysis: Scaled Sum of the Elements of the Correlation Matrix

It is clear from the results that Vortex outputs are as random as those of the SHA family across all variants of the family. These results also indicate that even with three rounds and degree of diffusion equal to 5 the Vortex family can output values with satisfactory statistical properties. This is due to the strong mixing performed by its compression function and the EMD transform used as domain extension which preserves the pseudo-random oracle and pseudo-random function properties.

Known Attacks

In what follows we provide a summary of how Vortex addresses a number of known attacks:

- **Algebraic attacks** may be applicable to Vortex. Vortex provides two mechanisms for mitigating algebraic attacks: First it sets the number of block cipher rounds to a tunable parameter, where the larger the number of rounds is the more complex an algebraic attack becomes. Second, it includes a degree of diffusion parameter for repeating the last sub-bock several times. These repetitions, together with other non-linearities of the block cipher push the complexity of algebraic attacks to a safety margin. For example, if the number of rounds is 3 and the degree of diffusion 5 (which is the default setup of the algorithm), each bit of the input goes through a SBox transformation 15 times. This is even stronger mixing as compared to AES 256 (14 times).
- **Related key attacks** are mitigated by using a Matyas-Meyer-Oseas structure which sets the attacker in control of the plaintext of the encryption but not the key.
- **Multi-collision attacks** may be applicable to Vortex; However the number of queries required for finding a single collision is quite high ($\sim 2^{122.5}$) as shown in the previous section and hence such attacks may not be practical.
- **Side channel attacks** can be mitigated using processor instructions that implement Rijndael rounds and carry-less multiplication using combinatorial logic as opposed to memory lookups.
- **Birthday attacks** are mitigated by feeding the same message word into two parallel block cipher stages and then mixing the results of the encryption using the Vortex merging function.

5. Performance Analysis

The main strength from using Vortex comes from the fact that the algorithm operates at a expected speed of 2.2-2.5 cycles per byte when using parameters N_R=3, M_T=0, D_F=5, and running in future processor architectures with instruction set support for Rijndael rounds and carry-less multiplication. Moreover, adding a single round to an A-Rijndael transformation increases the cost of the algorithm by no more than 0.5 cycles per byte.

An example a future processor architecture that will support such instructions, and which is familiar to the authors of this document, is Intel's next generation Core Micro-architecture. In this processor family, a new set of instructions will be introduced that enable high performance and secure round encryption and decryption. These instructions are AESENC (AES round encryption), AESENCLAST (AES last round encryption), AESDEC (AES round decryption) and AESDECLAST (AES last round decryption). The specification for these instructions is given in Table 6. Further information about the AES instructions can be found in the reference [10].

As shown in Table 6 the state of the cipher is kept at the destination XMM register (xmm1). The round key is kept at a source XMM register or can be obtained from memory. The AESENC instruction implements the following transformations of the AES specification in the order presented: Shift Rows, SBox, Mix Columns and Add Round Key. The AESENCLAST implements Shift Rows, SBox and Add Round Key but not

Mix Columns. The AESDEC instruction implements Inverse Shift Rows, Inverse SBox, Inverse Mix Columns and Add Round Key. Finally the AESDECLAST instruction implements Inverse Shift Rows, Inverse SBox, and Add Round Key but not Inverse Mix Columns.

Instruction	Description
AESENC xmm1, xmm2/128	performs one round of an AES encryption flow operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/128
AESENCLAST xmm1, xmm2/128	performs the last round of an AES encryption flow operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/128
AESDEC xmm1, xmm2/128	performs one round of an AES decryption flow using the equivalent inverse cipher operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/128
AESDECLAST xmm1, xmm2/128	performs the last round of an AES decryption flow using the equivalent inverse cipher operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/128

Table 6: AES instructions for round encryption and decryption

Instruction/ Description	
PCLMULQDQ xmm1, xmm2/m128, imm8 Carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, returning double quadwords. The immediate byte is used for determining which quadwords of xmm1 and xmm2/m128 should be used.	
imm8[7:0]	**Operation**
0x00	xmm2/m128[63:0] • xmm1[63:0]
0x01	xmm2/m128[63:0] • xmm1[127:64]
0x10	xmm2/m128[127:64] • xmm1[63:0]
0x11	xmm2/m128[127:64] • xmm1[127:64]

Table 7: The PCLMULQDQ instruction

Together with the AES instructions, Intel will also offer one new instruction supporting carry-less multiplication named PCLMULQDQ. The PCLMULQDQ instruction performs carry-less multiplication of two 64-bit words which are selected from the first

and the second operands according to the immediate byte value. The specification of the PCLMULQDQ instruction is given in Table 7.

In this document we argue that the introduction of such instructions will not characterize one particular processor family only, but eventually become a trend in the industry. There are several reasons for this: First, several hardware vendors besides Intel including Sun and IBM are researching the implementation of high performance AES encryption [16] and Galois Field multiplication [9] in hardware. Second, technologies such as composite fields are well known and can be used for constructing very compact AES implementations potentially exhibiting single clock throughput and small processing latency. Third, there is precedence with examples like the SSE instructions for multimedia processing, or the 64-bit extensions to integer arithmetic instructions which indicate that good technologies are eventually adopted by several semiconductor manufacturers. Fourth, even processors used in embedded systems already use hardware for AES encryption. Fifth, flexible crypto instructions like AES round instructions can be good hardware solutions for implementing a variety of algorithms. This is one of the aims of this work.

In this submission package we provide 4 implementations of the Vortex family: (i) a reference implementation; (ii) an optimized 64-bit implementation; (iii) an optimized 32-bit implementation; (iv) and an optimized assembly implementation with instruction 'stand-ins' for AESENC, AESENCLAST and PCLMULQDQ. The replacement we used for all three instructions is the integer multiplication instruction PMULUDQ. PMULUDQ demonstrates single clock throughput and three clock latency. From a researcher's perspective, we believe that such instruction can approximate the best performance that can come out of future implementations of AESENC, AESENCLAST and PCLMULQDQ. This statement reflects a belief based on well known characteristics of high performance and compact AES implementations and should not be seen as any form of product roadmap commitment.

Our performance results for the four types of implementation are shown in Table 8. Table 8 shows the performance of all members of the Vortex family when the number of rounds N_R is equal to 3, multiplication is carry-less ($M_T = 0$), and the degree of diffusion D_F is equal to 5. The results were obtained by hashing 1024 random messages of size 128 KB on a Core 2 Duo processor running at 3 GHz clock speed with 4 GB of RAM. Measurements were taken using the RDTSC instruction.

Implementation	Vortex 224 (cycles/byte)	Vortex 256 (cycles/byte)	Vortex 384 (cycles/byte)	Vortex 512 (cycles/byte)
Reference (64bit)	46.46	46.46	61.67	61.67
Optimized 64-bit	46.26	46.26	56.05	56.05
Optimized 32-bit	69.44	69.44	90.07	90.07
Assembly (stand-ins)	2.47	2.47	2.22	2.22

Table 8: Performance of the Vortex Family

Our optimized assembly code which results in the best performance is listed below:

```
Vortex_256_asm PROC

      ;rcx holds the pointer to the hash
      ;rdx holds the pointer to the data
      ;r8  holds the pointer to the rcon constants
      ;r9  holds the pointer to the length in blocks


      ;first we load the hash in the register pair <xmm1:xmm0>

      movdqu            xmm0, XMMWORD PTR [rcx]
      movdqu            xmm1, XMMWORD PTR [rcx+16]

      ;then we load the rcon constants in the registers xmm13, xmm14,
      ;xmm15

      movdqu            xmm13,      XMMWORD PTR [r8]
      movdqu            xmm14,      XMMWORD PTR [r8+16]
      movdqu            xmm15,      XMMWORD PTR [r8+32]

      ;xmm12 holds the constant zero
      pxor        xmm12,      xmm12


vortex_block_loop:
      mov               r10,  4
vortex_word_loop:
      ;first we load the data into the register xmm11
      movdqu            xmm11,      XMMWORD PTR [rdx]

      ;1st block: move the data to xmm4, xmm5
      movdqu            xmm4, xmm11
      movdqu            xmm5, xmm11

      ;do the modified key schedule
      paddq       xmm0, xmm13
      paddq       xmm1, xmm13

      ;permutation + sbox can be implemented using the future pshufb +
      ;aeseclast instructions
      ;we simulate these using pxor, pmuludq

      ;pshufb           xmm0
      pxor        xmm0, xmm13 ;1 clock stand-in
      ;pshufb           xmm1
      pxor        xmm1, xmm13
      ;aesenclast xmm0, xmm12
      pmuludq           xmm0, xmm12 ;3 clock latency, 1 clock
      ;throughput stand-in
      ;aesenclast xmm1, xmm12
      pmuludq           xmm1, xmm12

      ;the keys for the first round are in the registers xmm1:xmm0
      ;to pipeline the execution of the aes round instructions we begin
```

```
;preparing the key schedule for the next round

movdqu            xmm2, xmm0
movdqu            xmm3, xmm1
paddq       xmm2, xmm14
paddq       xmm3, xmm14
;pshufb           xmm2
pxor        xmm2, xmm14 ;1 clock stand-in
;pshufb           xmm3
pxor        xmm3, xmm14

;now we issue four aes instructions 2 for the first round and 2
;for the next key schedule
;aesenc           xmm4, xmm0
pmuludq           xmm4, xmm0
;aesenc           xmm5, xmm1
pmuludq           xmm5, xmm1
;aesenclast xmm2, xmm12
pmuludq           xmm2, xmm12
;3 clock latency, 1 clock throughput stand-in
;aesenclast xmm3, xmm12
pmuludq           xmm3, xmm12

;first round done - key schedule for second round prepared

;we begin preparing the key schedule for the third round

movdqu            xmm0, xmm2
movdqu            xmm1, xmm3
paddq       xmm0, xmm15
paddq       xmm1, xmm15
;pshufb           xmm0
pxor        xmm0, xmm15 ;1 clock stand-in
;pshufb           xmm1
pxor        xmm1, xmm15

;now we issue four aes instructions 2 for the first round and 2
;for the next key schedule
;aesenc           xmm4, xmm2
pmuludq           xmm4, xmm2
;aesenc           xmm5, xmm3
pmuludq           xmm5, xmm3
;aesenclast xmm0, xmm12
pmuludq           xmm0, xmm12 ;3 clock latency, 1 clock
;throughput stand-in
;aesenclast xmm1, xmm12
pmuludq           xmm1, xmm12

;second round done - key schedule for third round prepared

;aesenc           xmm4, xmm0
pmuludq           xmm4, xmm0
;aesenc           xmm5, xmm1
pmuludq           xmm5, xmm1

;last round done!
pxor        xmm4, xmm11 ;matyas-mayer-oseyas
```

```
pxor        xmm5, xmm11

;now we start the merging

movdqu              xmm1, xmm4
;pclmulqdq  xmm1, xmm5, 0x10 ;xmm1 holds the outer product
pmuludq             xmm1, xmm5
movdqu              xmm0, xmm4
;pclmulqdq  xmm0, xmm5, 0x01 ;xmm0 holds the inner product
pmuludq             xmm0, xmm5

movdqu              xmm2, xmm0
movdqu              xmm3, xmm1
shufpd              xmm0, xmm3, 0
shufpd              xmm1, xmm2, 0
pxor        xmm0, xmm4
paddq       xmm1, xmm5
;we are done!

add                 rdx,  16
dec                 r10
jnz                 vortex_word_loop
dec                 r9
jnz                 vortex_block_loop

;we load the hash back
movdqu              XMMWORD PTR [rcx], xmm0
movdqu              XMMWORD PTR [rcx+16], xmm1

RET
```

6. Selected Related Work and Acknowledgement

A lot of work has been done the recent years on the design of one way hash functions [3, 5, 6, 8, 11, 12, 15, 18, 19, 23]. In this section we acknowledge some seminal papers, which this design heavily draws from. First, this design uses the Enveloped Merkle-Damgård construction [3] as its domain extension transform to preserve the pseudo-random function and pseudo-random oracle properties besides collision resistance. Second, to avoid length extension attacks this design uses the concept of Merkle-Damgård strengthening presented in [8, 15]. Third, motivated by the weak security of the MDC2 mode, analyzed in [23] this design improves upon MDC2 by replacing its Feistel reordering stage by a more computationally complex merging function. Fourth the assembly implementation reported in this document uses processor instructions and development techniques discussed in [10, 24]. The authors would like to thank Jesse Walker and Gary Graunke for their useful discussions and comments on the algorithm specification and for their suggestions on how to demonstrate the security of the algorithm analytically and experimentally.

7. Concluding Remarks

We presented Vortex a new family of one way hash functions that can produce message digests of 224, 256, 384 and 512 bits. The main idea behind the design of these hash functions is that we use well known algorithms supporting very fast diffusion in a small number of steps. We presented a set of qualitative and analytical arguments why we believe Vortex is secure and described a set of experiments that gave us confidence that the Vortex design is not inferior to the SHA family in terms of its collision resistance and randomness of output differentials. Performance-wise the expected difference between Vortex and earlier work is expected to be substantial in future processor architectures. The Vortex family is expected to operate at a speed of less than 2.2-2.5 cycles per byte in future CPUs with instruction set support for Rijndael round computation and Galois Field (GF(2)) multiplication. We believe that the design of the Vortex family is important because it represents a scalable on-the-CPU solution for message and code integrity and can be used for supporting both high speed secure networking and protection against malware in next generation computing systems.

References

1. "Advanced Encryption Standard", Federal Information Processing Standards Publication 197, available at: http://csrc.nist.gov/publication/fips
2. K. Atasu, L. Breveglieri, and M. Macchetti, " Efficient AES implementations for ARM based platforms", *Proceedings of the 2004 ACM symposium on Applied Computing*, Nicosia, Cyprus, 2004.
3. M. Bellare and T. Ristenpart, "Multi-Property-Preserving Hash Domain Extension and the EMD Transform", *Advances in Cryptology – ASIACRYPT 2006*, LNCS 4284, pp. 299-314, 2006.
4. E. Biham, O. Dunkjelman and N. Keller, "Related key Impossible Differential Attacks on 8-Round AES-192", *Proceedings Cryptographer's Track, RSA Conference, RSA 2006,* San Jose, CA, 2006.
5. J. Black, M. Cochran and T. Shrimpton, "On the Impossibility of Highly Efficient Block Cipher-based Hash Functions", *Advances in Cryptology – EUROCRYPT 2005*, LNCS 3494, pp. 526-541, 2005.
6. J. S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function", *Crypto 2005*, Santa Barbara, CA, 2004.
7. J. Daemen and V. Rijman, "AES Proposal: Rijndael", available on-line at: http://www.cs.bilkent.edu.tr/~selcuk/teaching/cs519/rijndael.pdf
8. I. Damgård, "A Design Principle for Hash Functions", *Advances in Cryptology – CRYPTO 1989*, LNCS 435, pp. 416-427, 1989.
9. H. Eberle, N. Gura, and S. Chang-Shantz: "Cryptographic Processor for Arbitrary Elliptic Curves over GF(2^m)", ASAP 2003, 14th IEEE Int. Conference on

Application-specific Systems, Architectures and Processors, Hague, The Netherlands, June 24-26, 2003, pp. 444-454.

10. S. Gueron, "Advanced Encryption Standard (AES) Instruction Set", *available on-line at:* http://software.intel.com/sites/avx/
11. L. Knudsen, X. Lai and B. Preneel, "Attacks on Fast Double Block Length Hash Functions", *Journal of Cryptology*, No. 11, pp. 59-72, International Association for Cryptologic Research, 1998.
12. S. Lucks, "Design Principles for Iterated Hash Functions", *Cryptology ePrint Archive*, Report 2004/253, 2004. Available at: http://eprint.iacr.org
13. M. Maurer, R. Renner, C. Holenstein, "Indifferentiability, Impossibility Results on Reductions and Applications to the Random Oracle Methodology", *in TCC 2004*, Vol. 2951, LNCS pp.21-39, 2004.
14. A. Menezes, P. Oorschot and S. Vanstone, "Handbook of Applied Cryptography", *CRC Press,* 199
15. R. Merkle, "One Way Hash Functions and DES", *Advances in Cryptology – CRYPTO 1989*, LNCS 435, pp. 428-446, 1989.
16. S. Moriokah and A. Satoh, "An Optimized S-Box Circuit Architecture for Low Power RIJNDAEL Design", *Cryptographic Hardware and Embedded Systems - CHESS 2001*, pp. 172-186, 2002.
17. R. C.W. Phan, "Impossible Differential Cryptanalysis of 7 round Advanced Encryption Standard (AES)", *Information Processing Letters*, Vol. 91 (2004, pp. 33-38.
18. P. Rogaway and J. Steinberger, "Constructing hash Functions from Fixed-Key Block Ciphers", *Crypto 2008*, Santa Barbara, CA, 2008.
19. P. Rogaway and T. Shrimpton, "Cryptographic Hash-Functions Basics: Definitions, Implications, and Separations for Pre-image Resistance, Second Pre-image Resistance and Collision Resistance", *In Proceedings, Fast Software Encryption*, 2004.
20. A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao and P. Rohatgi, "Efficient Rijndael Encryption with Composite Field Arithmetic", *Cryptographic Hardware and Embedded Systems - CHESS 2001*, pp. 175-188, 2001.
21. A. Satoh, S. Moriokah, K. Takano and S. Munetoh, "A Compact Rijndael Hardware Architecture with SBox Optimization", *Advances in Cryptology – ASIACRYPT 2001*, LNCS 2248*,* pp. 239-254, 2001.
22. "Secure Hash Standard", Federal Information Processing Standards Publication 180-2, available at: http://csrc.nist.gov/publication/fips
23. J. P. Steinberger, "The Collision Intractability of MDC-2 in the Ideal Cipher Model", *Advances in Cryptology - EUROCRYPT 2007*, LNCS 4515, pp. 35-41, 2007.
24. S. Gueron and Michael E. Kounavis, "Carry-Less Multiplication and Its Usage for Computing The GCM Mode", *available on-line at:* http://software.intel.com/sites/avx/

Appendix A: Constant generation

Vortex uses a simple algorithm for constant generation. Two 32-bit values need to be stored by the algorithm in advance. These are a prime x and a modulus p listed below. All vortex constants result from x by iterating the formula $x \leftarrow (x^2 - x) \bmod p$. The ANSI C code implementing the generation of the Vortex constants is given below.

```
int i, j;
uint64_t p, x;
a0_b0_32_g = (uint8_t *)malloc(32);
a0_b0_64_g = (uint8_t *)malloc(64);
ta_tb_32_g = (uint8_t *)malloc(32);
ta_tb_64_g = (uint8_t *)malloc(64);

for(i=0; i < MAX_NUMBER_OF_ROUNDS; i++)
{
      rcon_256_g[i] = (uint8_t *)malloc(16);
      rcon_512_g[i] = (uint8_t *)malloc(32);
}

p = 4294967291;
x = 1414213562;
x = (x*x-x) % p;

for(i=0; i < 8; i++)
{
      ((uint32_t *)a0_b0_32_g)[i] = (uint32_t)x;
      x = (x*x-x) % p;
}
for(i=0; i < MAX_NUMBER_OF_ROUNDS; i++)
      for(j=0; j < 4; j++)
      {
            ((uint32_t *)(rcon_256_g[i]))[j] = (uint32_t)x;
            x = (x*x-x) % p;
      }
for(i=0; i < 16; i++)
{
      ((uint32_t *)a0_b0_64_g)[i] = (uint32_t)x;
      x = (x*x-x) % p;
}
for(i=0; i < MAX_NUMBER_OF_ROUNDS; i++)
      for(j=0; j < 8; j++)
      {
            ((uint32_t *)(rcon_512_g[i]))[j] = (uint32_t)x;
            x = (x*x-x) % p;
      }
for(i=0; i < 8; i++)
{
      ((uint32_t *)ta_tb_32_g)[i] = (uint32_t)x;
      x = (x*x-x) % p;
}
for(i=0; i < 16; i++)
{
```

```
      ((uint32_t *)ta_tb_64_g)[i] = (uint32_t)x;
      x = (x*x-x) % p;
}
return VORTEX_SUCCESS;
```

Appendix B: Test Vector with List of Intermediate Values

In what follows we provide a list of intermediate values for a known answer text vector for Vortex 256. The algorithm setup is the default (N_R=3, D_F=5, M_T=0). The input message is of length 2 and each bit value is equal to "1". The representation is little endian and the least significant bit of the quantities presented is the rightmost in the listing below.

```
message: 3
inputs to the vortex block compression function:
hash:
4932b0fc796d966ab6874438c48941ed435d25efe0c0c972766bdc05aa33ac12
input block:
987b3efa1d423a6c4ab885fbd29f4522531784875a34e4087b561a4d465dc66d
0000000000000002000000000000000000000000000000000000000000000007
before A-Rijndael
a:
435d25efe0c0c972766bdc05aa33ac12
b:
4932b0fc796d966ab6874438c48941ed
after A-Rijndael
a:
b5319fdd290ec6e34dd3218af5a01a8d
b:
2fbb92b3ddcdbe2e34e334e70c7052e5
after merging
a:
bf2d4faa78d3abb598d527f94354e4c2
b:
4efcf1a7b1003945513811237856bfcb
before A-Rijndael
a:
bf2d4faa78d3abb598d527f94354e4c2
b:
4efcf1a7b1003945513811237856bfcb
after A-Rijndael
a:
23dcb50e7ad740680f1ccb43b62318dc
b:
ef743f916f33d44cd2fc9677b5115ad2
after merging
a:
262123695ba4366d679762658efea08c
b:
0af217e7758318ba498a59ce92d27d62
before A-Rijndael
a:
262123695ba4366d679762658efea08c
b:
0af217e7758318ba498a59ce92d27d62
after A-Rijndael
a:
cb3e931feceef5029a8c5b12ac04ec99
b:
12605eb1adc372181ffba1ab5343b0a9
after merging
```

a:
c3969452a049470f9fd9f20a7fcef0cb
b:
1b3eb72abd82d6de030ea255deeb2001
before A-Rijndael
a:
c3969452a049470f9fd9f20a7fcef0cb
b:
1b3eb72abd82d6de030ea255deeb2001
after A-Rijndael
a:
d5f294b1db65cd6a58da00e7eb29a6b1
b:
9f4812d16567116b140b236c1a1e80bb
after merging
a:
ff4a179a60b1a3e19ebb28f3e59354bf
b:
ad573c421cbef91b8cda415b80ee85f6
before A-Rijndael
a:
ff4a179a60b1a3e19ebb28f3e59354bf
b:
ad573c421cbef91b8cda415b80ee85f6
after A-Rijndael
a:
f68220794a2654976c7d1f013165753f
b:
cdc3c4ed7c28a0819b2e43bb6b0115dc
after merging
a:
d930305fcdd8c54f806c4d54dfcc1feb
b:
40618fc43cb332028735a39ab900a09b
before A-Rijndael
a:
d930305fcdd8c54f806c4d54dfcc1feb
b:
40618fc43cb332028735a39ab900a09b
after A-Rijndael
a:
ec1641623b45e38438b60ffcb11468f7
b:
f54f5fb1857d2f9b8659bf95f46611c1
after merging
a:
fa2845b200385d2a3fa553ebea9b6c73
b:
69a0dd6348f118274ed81f43fcd3a912
before A-Rijndael
a:
fa2845b200385d2a3fa553ebea9b6c73
b:
69a0dd6348f118274ed81f43fcd3a912
after A-Rijndael
a:
bbf9bcf276ca962c28abb7c7d70f47f3

```
b:
8428284c2b6d0dfb8db57f779643e18d
after merging
a:
af0a9dbe26aa235f53fc2252c64a820f
b:
de5c789b89dd75217da01fa86aa9b62a
before A-Rijndael
a:
af0a9dbe26aa235f53fc2252c64a820f
b:
de5c789b89dd75217da01fa86aa9b62a
after A-Rijndael
a:
b0b7bfd9e47395c2b5254a54cc8fe39d
b:
1707f50a6ce2ca11d163cbd8116cc999
after merging
a:
b8ed7239b4d1c091d313587066e1376f
b:
96fe5bcb14dff7987297eb979e4691e6
final message digest:
96fe5bcb14dff7987297eb979e4691e6b8ed7239b4d1c091d313587066e1376f
```

Appendix C

Proof of lemma 1

A straightforward way to prove Lemma 1 is to build the truth table of the 64-bit carry-less multiplier and observe the frequency by which output values appear. Such approach is computationally infeasible since it requires storage space of at least 2^{102} GB. An alternative approach is to consider the carry-less multiplier as resulting from smaller input functions for which truth tables can be built.

We define the 'upper square' function $U_s(X,Y,i,j)$ as follows:

$$U_S(X,Y,i,j) = [u_s^{(7)}(X,Y,i,j)\ u_s^{(6)}(X,Y,i,j)...u_s^{(0)}(X,Y,i,j)] \tag{30}$$

where X, Y are given by (15), $i \in \{0,8,16,24,32,40,48\}$, $j \in \{8,16,24,32,40,48,56\}$ and the bit functions $u_s^{(0)}(X,Y,i,j)$,$u_s^{(1)}(X,Y,i,j),...,u_s^{(7)}(X,Y,i,j)$ are given by:

$$u_s^{(k)}(X,Y,i,j) = \bigoplus_{q=0}^{7} x_{i+q}y_{j+k-q} = x_i y_{j+k} \oplus x_{i+1}y_{j+k-1} \oplus ... \oplus x_{i+7}y_{j+k-7},\ 0 \le k \le 7 \tag{31}$$

Similarly we define the 'upper triangle' function $U_t(X,Y,i,j)$:

$$U_t(X,Y,i,j) = [u_t^{(7)}(X,Y,i,j)\ u_t^{(6)}(X,Y,i,j)...u_t^{(0)}(X,Y,i,j)] \tag{32}$$

where X, Y are given by (15), $i \in \{0,8,16,24,32,40,48,56\}$, $j = 0$ and the bit functions $u_t^{(0)}(X,Y,i,j)$,$u_t^{(1)}(X,Y,i,j),...u_t^{(7)}(X,Y,i,j)$ are given by:

$$u_t^{(k)}(X,Y,i,j) = \bigoplus_{q=0}^{k} x_{i+q}y_{j+k-q} = x_i y_{j+k} \oplus x_{i+1}y_{j+k-1} \oplus ... \oplus x_{i+k}y_j,\ 0 \le k \le 7 \tag{33}$$

Next, we define the 'lower square' function $L_s(X,Y,i,j)$ as follows:

$$L_S(X,Y,i,j) = [l_s^{(7)}(X,Y,i,j)\ l_s^{(6)}(X,Y,i,j)...l_s^{(0)}(X,Y,i,j)] \tag{34}$$

where X, Y are given by (15), $i \in \{1,9,17,25,33,41,49\}$, $j \in \{15,23,31,39,47,55,63\}$ and the bit functions $l_s^{(0)}(X,Y,i,j)$,$l_s^{(1)}(X,Y,i,j),...l_s^{(7)}(X,Y,i,j)$ are given by:

$$l_s^{(k)}(X,Y,i,j) = \bigoplus_{q=0}^{7} x_{i+k+q}y_{j-q} = x_{i+k} y_j \oplus x_{i+k+1}y_{j-1} \oplus ... \oplus x_{i+k+7}y_{j-7},\ 0 \le k \le 7 \tag{35}$$

Last we define the lower triangle' function $L_t(X,Y,i,j)$:

$$L_t(X,Y,i,j) = [l_t^{(6)}(X,Y,i,j)\ \ l_t^{(5)}(X,Y,i,j)...l_t^{(0)}(X,Y,i,j)] \tag{36}$$

where $X,\ Y$ are given by (15), $i = 57$, $j \in \{7,15,23,31,39,47,55,63\}$ and the bit functions $l_t^{(0)}(X,Y,i,j)$,$l_t^{(1)}(X,Y,i,j),...l_t^{(6)}(X,Y,i,j)$ are given by:

$$l_t^{(k)}(X,Y,i,j) = \bigoplus_{q=0}^{6-k} x_{i+k+q}y_{j-q} = x_{i+k}y_j \oplus x_{i+k+1}y_{j-1} \oplus ... \oplus x_{i+6}y_{j+k-6},,\ 0 \le k \le 6 \tag{37}$$

The importance of the functions defined by Eq. (30)-(37) lies on the fact that the 64-bit carry-less multiplication can be expressed as an exclusive OR (XOR) operation between their outputs. Let the output words W and Z be the byte sequences:

$$W = [W_7 : W_6 : ... : W_0], \quad Z = [Z_7 : Z_6 : ... : Z_0] \tag{38}$$

Then one can show that:

$$Z_k = (\bigoplus_{q=0}^{k-1} U_s(X,Y,\ 8\cdot q,\ 8\cdot(k-q))) \oplus U_t(X,Y,\ 8\cdot k,\ 0) \tag{39}$$

and:

$$W_k = (\bigoplus_{q=0}^{6-k} L_s(X,Y,\ 1+8\cdot(k+q),\ 63-8\cdot q)) \oplus L_t(X,Y,\ 57,\ 7+8\cdot q) \tag{40}$$

where $0 \le k \le 7$.

The functional decomposition of the 64-bit carry-less multiplier into upper, and lower, square and triangle functions is further shown in Figure 6.

We define 'adjacent' squares, as the outputs of upper or lower square functions of the form $U_s(X,Y,i,j)$ or $L_s(X,Y,i,j)$ for which the indexes i and j have the same sum and the indexes do not differ by more than 8 between different squares. Adjacent squares are illustrated as neighboring in Figure 6. For example, the squares $U_s(X,Y,0,16)$ and $U_s(X,Y,8,8)$ are adjacent. This is because the indexes 0, 16 and 8, 8 have the same sum and do not differ by more than 8. On the other hand the squares $U_s(X,Y,0,16)$ and $U_s(X,Y,0,24)$ are not adjacent. The concept of adjacency can be extended between square and triangle functions (upper or lower) in a similar manner.

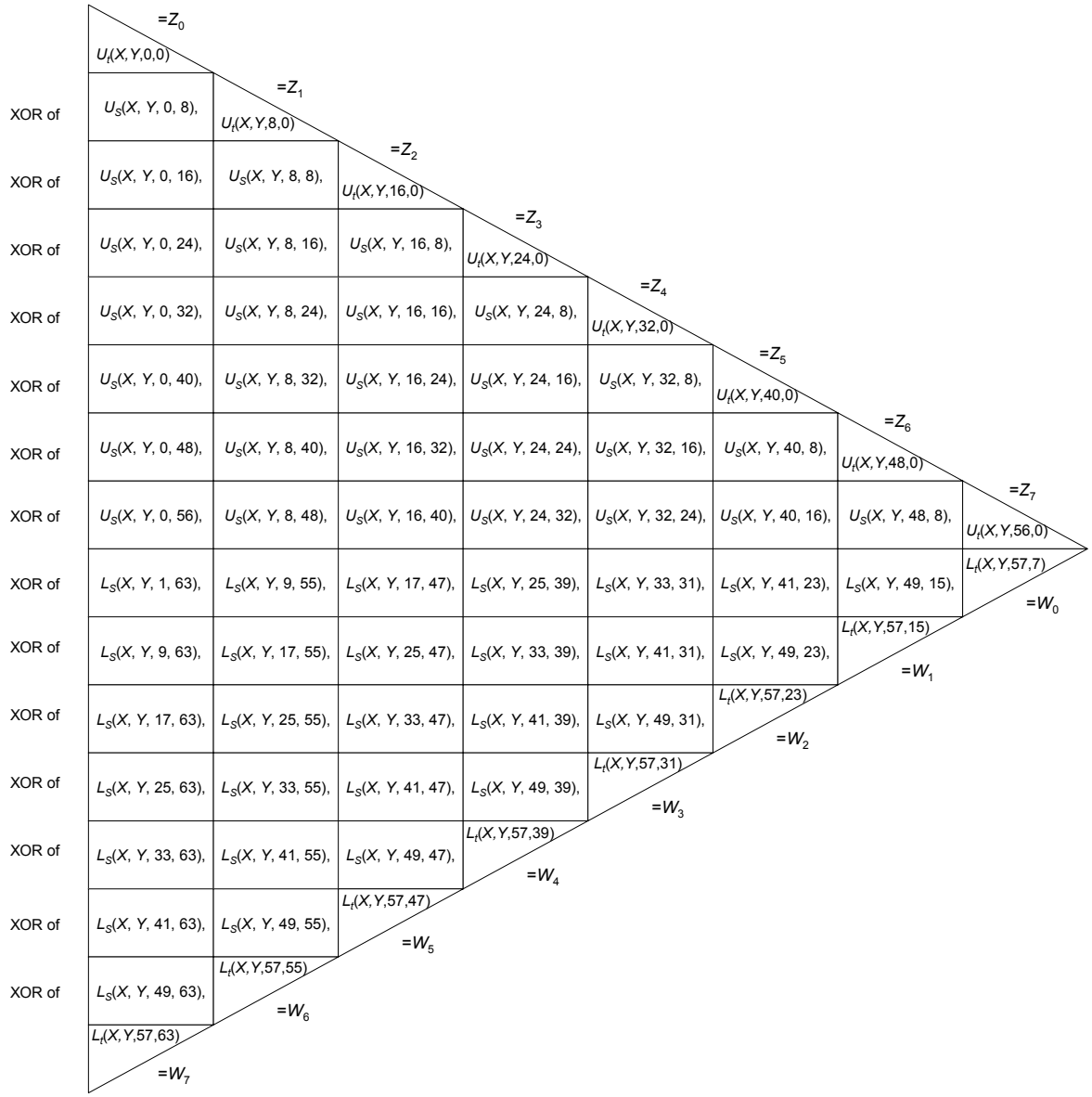

Figure 6: Functional Decomposition of a 64-bit Carry-less Multiplier

We observe that only adjacent squares or adjacent squares and triangles have associated input bits in common. Specifically, the squares $U_s(X,Y,I,J)$ and $U_s(X,Y,I+8,J-8)$ defined for some index pair I, J that satisfies (34) have seven input bits in common. These are the bits $[y_{J-7}y_{J-6}...y_{J-1}]$. Similarly squares $L_s(X,Y,I,J)$, $L_s(X,Y,I+8,J-8)$ defined for some index pair I, J that satisfies (23) have seven input bits in common. These are the bits $[x_{I+8}x_{I+9}...x_{I+14}]$. Adjacent upper squares $U_s(X,Y,I,J)$ and triangles $U_t(X,Y,I+8,J-8)$ have common input bits $[y_{J-7}y_{J-6}...y_{J-1}]$. Similarly adjacent lower squares $L_s(X,Y,I,J)$ and triangles $L_t(X,Y,I+8,J-8)$ have common input bits $[x_{I+8}x_{I+9}...x_{I+14}]$.

From Eq. (31) and (35) it is evident that the upper and lower square functions accept much fewer input bits as compared to the 64-bit carry-less multiplier. Each upper or lower square function accepts 23 input bits. An upper square accepts 8 bits from X and 15 bits from Y, where 7 bits from Y are in common with an adjacent square. A lower square accepts 15 bits from X and 8 bits from Y, where 7 bits from X are in common with an adjacent square.

Similarly, from Eq. (33) it is evident that the upper triangle function accepts 16 input bits, 8 bits from X and 8 bits from Y, where 7 bits from Y are in common with an adjacent square. From Eq. (37) it is also evident that a lower triangle function accepts 14 input bits, 7 bits from X and 7 bits from Y, where all 7 bits from X are in common with an adjacent square.

Based on these observations we build a truth table for each of these functions. The size of the truth table for both the upper and lower square functions is 8 MB. The size of the truth table for the upper triangle function is 64 KB. The size of the truth table for the lower triangle function is 16 KB. Using these truth tables we observe the frequency by which output values appear. Using these frequencies we compute probability distributions for each of these functions when the bits which are in common with adjacent squares are given.

In all expressions listed below it is assumed that the input to the carry-less multiplier is X and Y and it is uniformly distributed. We define the 'upper square –left' distribution as:

$$P^{(US-L)}(\tilde{v},\tilde{c}) = \Pr(U_s(X,Y,i,j) = \tilde{v} \wedge [y_{j-7}y_{j-6}...y_{j-1}] = \tilde{c}) \tag{41}$$

where the indexes i and j take the values defined in (34). Because of the uniformity of the square and triangle functions the probability distribution $P^{(US-L)}(\tilde{v},\tilde{c})$ is independent of the values of indexes i and j.

Similarly we define the 'upper square –right' distribution as:

$$P^{(US-R)}(\tilde{v},\tilde{c}) = \Pr(U_s(X,Y,i,j) = \tilde{v} \wedge [y_{j+1}y_{j+2}...y_{j+7}] = \tilde{c}) \tag{42}$$

the 'lower square –left' distribution as:

$$P^{(LS-L)}(\tilde{v},\tilde{c}) = \Pr(L_s(X,Y,i,j) = \tilde{v} \wedge [x_{i+8}x_{i+9}...x_{i+14}] = \tilde{c}) \tag{43}$$

and the 'lower square-right distribution as:

$$P^{(LS-R)}(\tilde{v},\tilde{c}) = \Pr(L_s(X,Y,i,j) = \tilde{v} \wedge [x_i x_{i+1}...x_{i+6}] = \tilde{c}) \tag{44}$$

Last we define the 'upper triangle distribution' as:

$$P^{(UT)}(\widetilde{v},\widetilde{c}) = \Pr(U_t(X,Y,i,j) = \widetilde{v} \wedge [y_{j+1}y_{j+2}...y_{j+7}] = \widetilde{c}) \tag{45}$$

and the 'lower triangle distribution' as:

$$P^{(LT)}(\widetilde{v},\widetilde{c}) = \Pr(L_t(X,Y,i,j) = \widetilde{v} \wedge [x_i x_{i+1}...x_{i+6}] = \widetilde{c}) \tag{46}$$

The probability distributions defined in Eq. (42)-(46) are independent of the choices of i, j. If the inputs X and Y are uniformly distributed, the probability distributions (41)-(46) can be computed from the truth tables of the upper and lower, square and triangle functions.

We continue with the proof of Lemma 1 by computing the probability distribution of the exclusive OR (XOR) between the outputs of two adjacent squares - or one triangle and its adjacent square. We define 'upper adjacent square XOR' distribution as:

$$P^{(US-A)}(\widetilde{v}) = \Pr(U_s(X,Y,i,j) \oplus U_s(X,Y,i+8,j-8) = \widetilde{v}) \tag{47}$$

The probability distribution (47) is obtained from the upper square left (41) and right (42) distributions as follows:

$$P^{(US-A)}(\widetilde{v}) = \sum_{\widetilde{c}\in[0,127],\ \widetilde{u},\widetilde{w}\in[0,255],\ \widetilde{u}\oplus\widetilde{w}=\widetilde{v}} P^{(US-L)}(\widetilde{u},\widetilde{c}) \cdot P^{(US-R)}(\widetilde{w},\widetilde{c}) \tag{48}$$

Similarly, we define the 'lower adjacent square XOR' distribution:

$$P^{(LS-A)}(\widetilde{v}) = \Pr(L_s(X,Y,i,j) \oplus L_s(X,Y,i+8,j-8) = \widetilde{v}) \tag{49}$$

which is computed from the lower square left (43) and right (44) distributions:

$$P^{(LS-A)}(\widetilde{v}) = \sum_{\widetilde{c}\in[0,127],\ \widetilde{u},\widetilde{w}\in[0,255],\ \widetilde{u}\oplus\widetilde{w}=\widetilde{v}} P^{(LS-L)}(\widetilde{u},\widetilde{c}) \cdot P^{(LS-R)}(\widetilde{w},\widetilde{c}) \tag{50}$$

We also define the 'upper adjacent square-triangle XOR' distribution as:

$$P^{(UST-A)}(\widetilde{v}) = \Pr(U_s(X,Y,i,j) \oplus U_t(X,Y,i+8,j-8) = \widetilde{v}) \tag{51}$$

which is computed from the upper square left (41) and triangle (45) distributions:

$$P^{(UST-A)}(\widetilde{v}) = \sum_{\widetilde{c}\in[0,127],\ \widetilde{u},\widetilde{w}\in[0,255],\ \widetilde{u}\oplus\widetilde{w}=\widetilde{v}} P^{(US-L)}(\widetilde{u},\widetilde{c}) \cdot P^{(UT)}(\widetilde{w},\widetilde{c}) \tag{52}$$

Last we define the 'lower adjacent square-triangle XOR' distribution as:

$$P^{(LST-A)}(\tilde{v}) = \Pr(L_s(X,Y,i,j) \oplus L_t(X,Y,i+8,j-8) = \tilde{v}) \tag{53}$$

which is computed from the lower square left (41) and triangle (45) distributions:

$$P^{(LST-A)}(\tilde{v}) = \sum_{\tilde{w},\tilde{c}\in[0,127],\ \tilde{u}\in[0,255],\ \tilde{u}\oplus\tilde{w}=\tilde{v}} P^{(LS-L)}(\tilde{u},\tilde{c}) \cdot P^{(LT)}(\tilde{w},\tilde{c}) \tag{54}$$

The probability distributions computed from (47), (49), (51), (53) are used together with Eq. (39) and Eq. (40) to compute the probability distribution of the output of the 64-bit carry-less multiplier, assuming that the input is uniformly distributed.

Let the byte representation of a 128-bit value $[\tilde{W} : \tilde{Z}]$ be:

$$\tilde{W} = [\tilde{W}_7 : \tilde{W}_6 : \ldots : \tilde{W}_0], \quad \tilde{Z} = [\tilde{Z}_7 : \tilde{Z}_6 : \ldots : \tilde{Z}_0] \tag{55}$$

The probability that the 64-bit carry-less multiplier output $[W : Z]$ is equal to $[\tilde{W} : \tilde{Z}]$ is given by the product:

$$\Pr([W : Z] = [\tilde{W} : \tilde{Z}]) = (\prod_{k=0}^{7} \Pr(W_k = \tilde{W}_k)) \cdot (\prod_{k=0}^{7} \Pr(Z_k = \tilde{Z}_k)) \tag{56}$$

where the output words W and Z are expressed as byte sequences as in Eq. (38).

Each of the output byte probabilities $\Pr(Z_k = \tilde{Z}_k)$ for k =0 , 1,…,7 are computed from the XOR probability distributions for adjacent squares and adjacent squares/triangles using the algorithm ‘compute $\Pr(Z_k = \tilde{Z}_k)$ ’ shown below.

compute $\Pr(Z_k = \tilde{Z}_k)$
begin
    if (*k=0***) return** $\sum_{\tilde{c}\in[0,127]} P^{(UT)}(\tilde{Z}_0,\tilde{c})$ // the output results from a single triangle
    else // the output results from XOR-ing squares and a triangle
        for $i \leftarrow 0$ **to** 255 **do** $T_2[i] \leftarrow P^{(UST-A)}(i \oplus \tilde{Z}_k)$
        for $i \leftarrow 0$ **to** k-1 **do**
        begin
            $T_1 \leftarrow T_2$
            for $j \leftarrow 0$ **to** 255 **do** $T_2[j] \leftarrow \sum_{\tilde{u},\tilde{v}\in[0,255],\tilde{u}\oplus\tilde{v}=j} P^{(US-A)}(\tilde{u}) \cdot T_1[\tilde{v}]$
        end
        return $\sum_{i=0}^{255} (\sum_{\tilde{c}\in[0,127]} P^{(US-L)}(i,\tilde{c})) \cdot T_2[i])$
end

The rationale behind this algorithm is that the event $Z_k = \widetilde{Z}_k$ is expressed as the union of other events. These are the events that certain values, from 0 to 255, appear at the outputs of the squares that contribute to the computation of $Z_k = \widetilde{Z}_k$. Knowing the XOR probability distributions for adjacent squares and adjacent squares/triangles helps us compute the probabilities of such events. The computation starts from the square that is adjacent to the upper triangle resulting in $Z_k = \widetilde{Z}_k$ and proceeds by taking one additional adjacent square into account at a time. The computation stops when all squares that contribute to $Z_k = \widetilde{Z}_k$ have been taken into account. The resulting probability is returned. Similarly the output byte probabilities $\Pr(W_k = \widetilde{W}_k)$ for k =0 , 1,…,7 are computed using the algorithm 'compute $\Pr(W_k = \widetilde{W}_k)$'

compute $\Pr(W_k = \widetilde{W}_k)$
begin
  if (k=7**) return** $\sum_{\widetilde{c}\in[0,127]} P^{(LT)}(\widetilde{W}_7, \widetilde{c})$ // the output results from a single triangle
  else // the output results from XOR-ing squares and a triangle
    for $i \leftarrow 0$ **to** 255 **do** $T_2[i] \leftarrow P^{(LST-A)}(i \oplus \widetilde{W}_k)$
    for $i \leftarrow 0$ **to** 6-k **do**
    begin
      $T_1 \leftarrow T_2$
      for $j \leftarrow 0$ **to** 255 **do** $T_2[j] \leftarrow \sum_{\widetilde{u},\widetilde{v}\in[0,255],\widetilde{u}\oplus\widetilde{v}=j} P^{(LS-A)}(\widetilde{u}) \cdot T_1[\widetilde{v}]$
    end
    return $\sum_{i=0}^{255}(\sum_{\widetilde{c}\in[0,127]} P^{(LS-L)}(i,\widetilde{c})) \cdot T_2[i])$
end

k	**upper bound for** $\Pr(Z_k = \widetilde{Z}_k)$	**upper bound for** $\Pr(W_k = \widetilde{W}_k)$
0	$2^{-5.68}$	2^{-8}
1	$2^{-7.98}$	2^{-8}
2	2^{-8}	2^{-8}
3	2^{-8}	2^{-8}
4	2^{-8}	2^{-8}
5	2^{-8}	2^{-8}
6	2^{-8}	2^{-8}
7	2^{-8}	$2^{-4.83}$

Table 9: Upper bounds for the byte probabilities $\Pr(W_k = \widetilde{W}_k)$ **and** $\Pr(Z_k = \widetilde{Z}_k)$

Upper bounds for the probabilities $\Pr(Z_k = \widetilde{Z}_k)$ and $\Pr(W_k = \widetilde{W}_k)$ are computed using the algorithms presented above and the truth tables for the upper and lower square and triangle functions. These upper bounds are shown in Table 9. From the data of Table 9 and Eq. (56) Lemma 1 is proven.