ESSENCE: A Family of Cryptographic Hashing Algorithms

Jason Worth Martin

October 21, 2008

Abstract

This paper describes the compression functions for a family of Merkle-Damgård based cryptographic hashing algorithms.

The compression functions are based on a nonlinear, key-dependent permutation, E, of 256 or 512 bits built from 32 or 64 eight-bit nonlinear feedback shift-registers run in parallel with linear mixing between the shift-registers.

The E permutation has been designed so that it can execute completely within the register file of modern 64-bit microprocessors and in constant time, thus increasing its resistance to side-channel attacks.

We give a complete description of all criteria used for the constructions and provide differential and linear cryptanalysis.

1 Introduction

This paper describes a family of compression functions designed to be used to construct cryptographic hashing algorithms. The compression functions are intended to be used within a Merkle hash tree (similar to the Tiger hash tree) where the lowest level blocks are hashed with a Merkle-Damgård construction. As such constructions are dealt with elsewhere in cryptographic literature, this paper focuses on the ESSENCE compression functions. As Merkle hash tree algorithms are inherently parallel, we will limit our discussion of parallel speedups to the instruction level parallelism within the compression function itself.

Since many real-world successful attacks against cryptosystems exploit vulnerabilities in the implementation rather than the theoretical underpinnings of the system, the ESSENCE compression functions have been designed with several pragmatic, as well as mathematical, criteria in mind. For example, Daniel J. Bernstein demonstrated in [Ber04] that standard server-side AES implementations were vulnerable to timing attacks even though the AES evaluation committee had considered timing attacks impractical. Bernstein's work showed that what made AES susceptible was its reliance on data-dependent look-up tables and that any implementation of AES which did not rely on look-up tables would be unacceptably slow. Since cryptographic hashing algorithms are

often employed for tasks such as key generation or message authentication codes on publicly accessible servers, we wish to protect against timing attacks. So, one of the principal design criteria for ESSENCE algorithms is that it should be possible to construct implementations which have constant execution times (meaning that data of the same size are processed in the same time) which are reasonably fast. As such, ESSENCE algorithms do not require any look-up tables or operations whose timing might be data-dependent for reasonable implementations.

Likewise, to help protect the internals of the executing algorithms from side-channel attacks such as power analysis, fault analysis, memory leaks, etc. on a multi-user system (such as a server), it seems prudent to design an algorithm where the compression function can be performed completely within the processor's register file. The ESSENCE compression functions can indeed be implemented completely within the CPU registers of a modern 64-bit processor (such as the x86_64 family). There is no requirement for a "key-setup" or "input-expansion" phase involved in computing the compression function.

Another important design consideration is to make the algorithm easy to implement in systems with limited processing capability (e.g. 8-bit microprocessors). To this end, the operations in the compression function are limited to bit-wise AND, NOT, XOR, and SHIFT.

All of these design constraints do force a performance trade off. For example, we have written naive C code implementations of the E permutation with and without look-up tables. The implementation without the look-up tables runs at approximately 160 processor cycles/byte (on an x86_64 Linux platform), while the implementation with look-up tables executes at approximately 53 cycles/byte on the same platform. (We expect that highly tuned assembly code would perform better, but at the time of this writing that has not been implemented.) So, there is certainly a trade-off for resistance to timing attacks. However, that trade-off is not severe for ESSENCE, and the existence of demonstrated real-world successful attacks against the timing-sensitive implementations of otherwise secure cryptosystems leads us to believe that it is necessary to have an algorithm which can have reasonable implementations that are resistant to side-channel attacks.

2 Notation and Numbering Conventions

The notation used within this document should be familiar to most C programmers. This section is intended for the mathematical audience who may be unfamiliar with the C programming language.

Unless otherwise stated, through this document we will attempt to adhere to the following conventions:

1. Bits and bytes.

   We use the term "bit" to denote a single binary digit (e.g. a value of either 0 or 1). We use the term "byte" to denote an 8-bit unsigned integer

(taking values between 0 and 255 inclusive).

2. Use of C notation for hexadecimal constants.

   We use the prefix `0x` to denote a hexadecimal number, and we use the letters `a,b,...,f` to denote the values `10,11,...,15` in hexadecimal. For example, we write `0xabcd1234` to denote the hexadecimal constant `abcd1234` which is equal to the decimal number 2882343476.

3. Indexing begins at 0 instead of 1.

   We begin all of our indexing at 0 instead of 1. This choice is to make all of our notation consistent with the C programming language.

4. Little Endian bit, byte, and block numbering.

   We use the principle of "less significant components receive a smaller index" when indexing the components of any regular structure. For example, we shall say that the integer `0xabcd1234` has byte 0 equal to `0x34` and byte 3 equal to `0xab`, and it has bit 0 equal to 0 and bit 31 equal to 1.

   We follow this principle when coercing a string of bits to be a polynomial in $\mathbb{F}_2[x]$. For example, we may regard `0x87` in $\mathbb{F}_2[x]$ as the polynomial $x^7 + x^2 + x + 1$ since we assign the least significant bit to be the coefficient of the smallest power of x.

   Note that our convention refers to the index of a component, not necessarily its position on the printed page.

5. Use of C array notation for indexing components of regular structures.

   We will use the C array notation (e.g. `someobject[i]`) for indexing the components of any regular structure. For example, let `r0` be a 64-bit integer, then we might use `r0[0]` to denote the least significant bit of `r0`, and `r0[63]` to denote its most significant bit.

   Unless otherwise stated, we use the array index notation to denote the smallest natural components of the object. Therefore `r0[0]` is the least significant bit of `r0`, not its least significant byte. Should we wish to discuss the bytes of `r0`, we will explicitly state the change in notation.

3 The E Permutation

The E permutation is constructed from the update function represented in Figure 1. In the diagram, $r0, r1, \ldots, r7$ represent 8 unsigned integers each consisting of either 32 or 64 bits depending upon the hash size desired. For hashes of 256 bits or less, $r0, r1, \ldots, r7$ are all 32-bit integers. For hashes of more than 256 bits (up to 512 bits), $r0, r1, \ldots, r7$ are 64-bit integers. The variable labeled k is likewise an integer of the same size as $r0, r1, \ldots, r7$.

In Figure 1 the symbol $\oplus$ represents bit-wise xor (i.e. bit-wise "exclusive or" or simply vector addition when the inputs are regarded as vectors over $\mathbb{F}_2$).

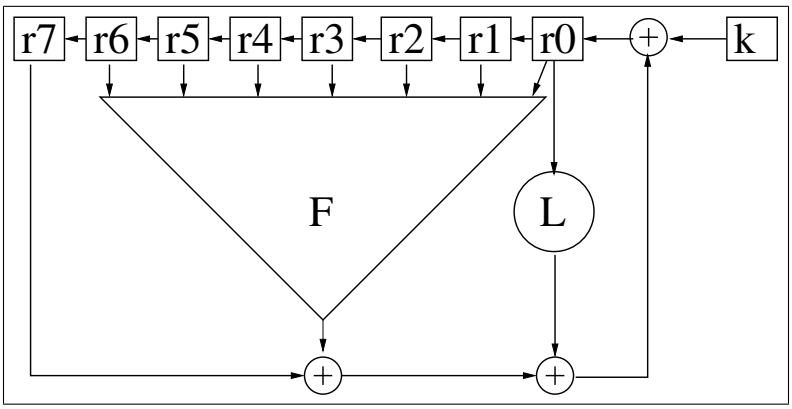

Figure 1: E Permutation Logic Diagram

```
tmp := F(r6,r5,r4,r3,r2,r1,r0) xor r7 xor L(r0)
r7 := r6
r6 := r5
r5 := r4
r4 := r3
r3 := r2
r1 := r1
r1 := r0
r0 := tmp xor k
```

Figure 2: Pseudo-code for E Permutation Logic

The arrows in the diagram represent assignment or the inputs to functions. The triangle labeled "F" represents a bit-wise application of the function expressed in equation (1) and the circle labeled with an "L" represents a linear function expressed in equation (4). The diagram represents a single "step" used for updating the values of $r0, r1, \ldots, r7$. Figure 2 gives the pseudo-code for a single step of the E permutation logic.

For each step of the update logic the value of `k` may vary (although in some instances it may be fixed to zero). The number of times the update logic is stepped is implementation specific, however the recommended number of steps is 24 and it should always be a multiple of 8. This is justified in sections 3.5 and 3.6

3.1 Motivation for Design of E

In [Sie84] Siegenthaler shows that if a Boolean function, $\alpha(x_1, \ldots, x_n)$, is correlation-immune, then it is essentially just an affine combination of the inputs. So if the

values of the x_i were to come from linear sources, such as linear feedback shift registers, then the result of combining the x_i using a correlation-immune α would be a completely linear system, and hence vulnerable to well known linear methods. If α is taken to be non-linear, then it becomes more vulnerable to correlation attacks.

So, our design approach is to use nonlinear feedback to drive the shift registers. We design the linear combining function, L, to maximize resistance to correlation attacks, while maximizing the linear complexity of the registers to maximize the resistance to linear analysis.

3.2 F – The Feedback Function

Let $\mathbb{F}_2$ denote the field of two elements. We define the feedback function, F, as the Boolean map $\mathrm{F} : \mathbb{F}_2^7 \to \mathbb{F}_2$ given by

$$
\begin{aligned}
\mathrm{F}(a,b,c,d,e,f,g) \quad = \quad & abcdefg + abcdef + abcefg + acdefg + \\
& abceg + abdef + abdeg + abefg + \\
& acdef + acdfg + acefg + adefg + \\
& bcdfg + bdefg + cdefg + \\
& abcf + abcg + abdg + acdf + adef + \\
& adeg + adfg + bcde + bceg + bdeg + cdef + \\
& abc + abe + abf + abg + acg + adf + \\
& adg + aef + aeg + bcf + bcg + bde + \\
& bdf + beg + bfg + cde + cdf + def + \\
& deg + dfg + \\
& ad + ae + bc + bd + cd + \\
& ce + df + dg + ef + fg + \\
& a + b + c + f + 1
\end{aligned}
\tag{1}
$$

where the multiplication and addition are taken in $\mathbb{F}_2$ (e.g. the addition is the same as bitwise "xor" and the multiplication is the same as bit-wise "and").

Any Boolean function may be expressed uniquely, up to rearrangement, as a minimal sum of products over $\mathbb{F}_2$ (often called the "algebraic normal form"). Equation (1) is such a representation for F. However, it is not a computationally efficient representation. Appendix A gives better representations of F for both hardware and software implementations. For clarity, and to allow for error checking, Tables 2 through 5 in appendix D give the explicit value of F for each possible input.

A note of caution: in computer science and engineering literature it is common to see the addition symbol, +, used to denote "inclusive or" in a "sum of min-terms" presentation for a Boolean function. That is not the case in equation (1).

3.2.1 Selection Criteria for F

As F is the only nonlinear function in the ESSENCE algorithms, the security of the algorithms is heavily dependent on F. We first consider the F function

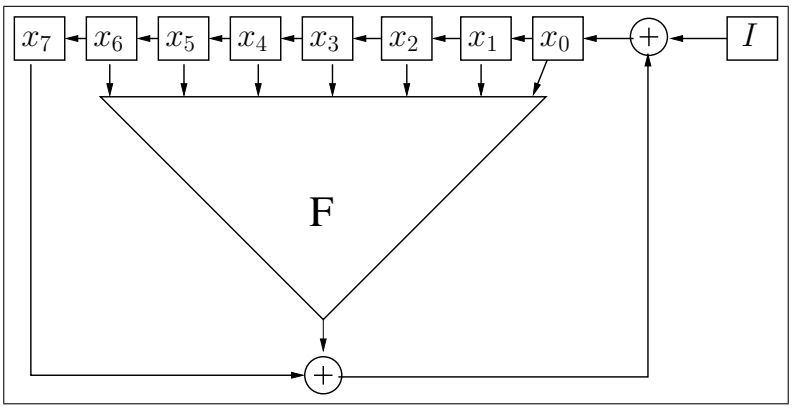

Figure 3: F Based Shift Register

used as the feedback function for a single shift register. In Figure 3 the stages of the shift register are labeled as $x_7, x_6, \ldots, x_0$ and I is the input. The stages $x_7, x_6, \ldots, x_0$ and I are all single bits.

Solomon Golomb demonstrates in [Gol67] (ch. VI §2 Theorem 1) that a maximal length shift register sequence on n stages is obtained from a feedback function on the first $n-1$ stages added to the last stage as shown in Figure 3, which explains why F takes only seven inputs instead of eight. The F function used in ESSENCE was found using a pseudo-random search with respect to the following criteria (in order of importance).

1. The shift register sequence formed by F must be a de Bruijn sequence.

   A de Bruijn sequence for an n-bit shift register sequence is a sequence generated by an n-bit shift register which contains all possible 2^n values that the register can hold. This is equivalent to the sequence being of maximal length for the register.

   The function F produces a de Bruijn sequence from the shift register illustrated in Figure 3 with I set to zero.

2. The shift register sequence formed by F must have maximal linear complexity.

   Given any sequence, the linear complexity of the sequence is the length of the smallest linear feedback shift register (LFSR) which can generate the sequence. The maximal linear complexity for a shift register sequence generated by an n-bit register is 2^n.

   The function F, used in the configuration shown in Figure 3 with I set to zero produces a shift register sequence with maximal linear complexity (i.e. its linear complexity is 256 with the connecting polynomial $x^{256}+1$).

3. The shift register sequence formed by F minimizes the differential characteristic.

   Let $h_{\mathrm{F},m} : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ be defined as follows. Let $a \in \mathbb{F}_2^8$ be an 8-bit value. Place the bits of a in the shift register in Figure 3 with $x_7 = a[7], \ldots, x_0 = a[0]$. Hold I to zero and step the register m times. Let $b \in \mathbb{F}_2^8$ be the resulting value of the register (i.e. $b[7] = x_7, \ldots, b[0] = x_0$). We define $h_{\mathrm{F},m}(a) = b$. For all values of m, $h_{\mathrm{F},m}$ is a bijection.

   We may now construct the differential table for $h_{\mathrm{F},m}$ as described in [BS91]. The rows of the table correspond to the differences in the input (Δ_{in}) and the columns correspond to the differences in the output (Δ_{out}). The entry in each table position corresponds to the total number of times that a given input difference yields a given output difference as we run through all possible pairs with distinct differences.

   As we are dealing with functions on $\mathbb{F}_2^8$, the differential table has $(255)(255) = 65025$ non-trivial entries. The number of distinct possible pairs of inputs yielding a non-trivial difference is $\frac{(256)(255)}{2} = 32640$ (we divide by 2 because symmetric pairs yield the same difference). Ideally, then, we would like the 32640 possible results to be distributed close to uniformly in the differential table, and we wish to avoid having any single entry in the table be particularly large.

   In our search algorithm for F we begin with a candidate feedback function, F^*, and for $1 \leq m \leq 16$ we construct the differential characteristic table for $h_{\mathrm{F}^*,m}$. For each of these tables we find the maximal non-trivial entry in the differential characteristic table. The m corresponding to the table with the smallest maximal non-trivial entry is called the *best stepping* for F^*. We call the maximal non-trivial entry in the differential characteristic table corresponding to the best stepping the *differential characteristic* of F^*. Given two candidate functions to compare, we select the one with the smallest differential characteristic. If the differential characteristics are equal, we select the function whose best stepping is smallest.

   The function, F, that was found has 8 as its best stepping and a differential characteristic of 4.

4. The shift register sequence formed by F minimizes the stream differential characteristic.

   Consider the shift register described by Figure 3 with a fixed input stream of k-bits (entering the register via I). We label the stream as $I_0, I_1, \ldots, I_{k-1}$, and consider the what happens as the register is stepped k times with $I = I_0$ on the first step, $I = I_1$ on the second step, etc. This provides us with a key-dependent permutation on $\mathbb{F}_2^8$ with the key being $(I_0, I_1, \ldots, I_{k-1})$.

   Let $a \in \mathbb{F}_2^8$ be an 8-bit value. We define the function $h_{\mathrm{F}}(a; I_0, \ldots, I_{k-1})$ by placing the bits of a in the shift register in Figure 3 with $x_7 = a[7], \ldots, x_0 = a[0]$. Put $I = I_0$ and step the register once. Put $I = I_1$ and

step the register again. Continue stepping the register with $I = I_{i-1}$ for the i^{th} step until the register has been stepped k times. Let $b \in \mathbb{F}_2^8$ be the resulting value of the register (i.e. $b[7] = x_7, \ldots, b[0] = x_0$). We define $h_{\mathrm{F}}(a; I_0, I_1, \ldots, I_{k-1}) = b$.

We now define the *stream differential table* by running through all 2^k possible values for $(I_0, \ldots, I_{k-1})$ for each distinct input pair and computing the output difference.

More precisely, let Δ_{in} be a non-trivial input difference. Let (a, a') run through the 128 distinct pairs of inputs with input difference equal to Δ_{in} and let $(I_0, \ldots, I_{k-1})$ run through all 2^k possible values. Compute $\Delta_{\text{out}} = h_{\mathrm{F}}(a; I_0, \ldots, I_{k-1}) \oplus h_{\mathrm{F}}(a'; I_0, \ldots, I_{k-1})$ for each case. The stream differential table, like the the differential table described in the previous section, consists of entries indicating the total number of times that a given input difference yielded a given output difference. (However, due to the additional 2^k applications of the function, the stream differential table can be much closer to uniformly distributed than the differential table.)

In our search algorithm for F we begin with a candidate feedback function, F^*, and set $k = 16$ (computational limits prevented analysis of larger streams). We construct the stream differential characteristic table for h_{F^*}. Since there are $128 \cdot 255$ possible non-trivial distinct input pairs, 2^k possible key values, and $255 \cdot 255$ possible non-trivial entries in the table, the mean value of the entries in the table is

$$\eta = \frac{128 \cdot 255 \cdot 2^{16}}{255 \cdot 255} \approx 32896.5.$$

Since we want to ensure the the table is close to uniformly distributed, we want to ensure that each entry is close to the mean value. So, we define the *stream differential characteristic* of F^* to be the greatest absolute difference of any non-trivial table entry from the mean. Given two candidate functions that are equal with respect to the previous criteria, we take the one with the smallest stream differential characteristic.

The function, F, that was selected has a stream differential characteristic of 6911.498.

5. The Boolean function F maximizes Hamming distance from any affine approximation.

   Given two candidate functions which are equal with respect to the previous criteria, we choose the one which has the greatest Hamming distance from any affine approximation. For the chosen F, the best possible affine approximation agrees with the function for 79 out of 128 possible entries.

3.3 L – The Linear Function

The linear function L is defined differently depending upon the size of the input (32-bit versus 64-bit). In both cases L is defined in terms of a generating

polynomial in $\mathbb{F}_2[x]$, and may easily be implemented via a linear feedback shift register in Galois configuration. We give the mathematical definition here and discuss the shift register implementation in appendix B.

Let $p_{64} \in \mathbb{F}_2[x]$ be given by

$$
\begin{aligned}
p_{64}(x) \quad = \quad & x^{64} + x^{63} + x^{61} + x^{60} + x^{55} + x^{53} + x^{50} + x^{49} + \\
& x^{46} + x^{44} + x^{41} + x^{40} + x^{36} + x^{33} + x^{32} + \\
& x^{31} + x^{30} + x^{29} + x^{26} + x^{25} + x^{23} + x^{20} + x^{18} + x^{17} + \\
& x^{14} + x^{13} + x^{11} + x^{8} + x^{7} + x^{4} + x^{2} + x + 1
\end{aligned} \tag{2}
$$

and let $C_{p_{64}}$ be the companion matrix of $p_{64}(x)$. (The polynomial p_{64} is *primitive*, which means that it is irreducible and that $\bar{x}$, the natural image of x from $\mathbb{F}_2[x]$ into the field $\mathbb{F}_2[x]/(p_{64}(x))$, is a generator of the multiplicative group of $\mathbb{F}_2[x]/(p_{64}(x))$.) Recall that the companion matrix of a polynomial $f(x) = b_n x^n + b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$ is the $n \times n$ matrix

$$
C_f = \begin{pmatrix}
0 & 1 & 0 & 0 & \dots & 0 \\
0 & 0 & 1 & 0 & \dots & 0 \\
 & & & \ddots & & \\
 & & & & & \\
0 & 0 & 0 & 0 & \dots & 1 \\
-b_{n-1} & -b_{n-2} & -b_{n-3} & -b_{n-4} & \dots & -b_0
\end{pmatrix}.
$$

Now let $v \in \mathbb{F}_2^{64}$ be a row vector, then we define

$$L_{64}(v) = vC_{p_{64}}^{64}.$$

Recall our little endian convention that less significant coefficients correspond to smaller indices. So, if `r0` is a 64-bit integer, then to compute L_{64}(`r0`) we treat the least significant bit of `r0` as the first component of the vector in $\mathbb{F}_2^{64}$, the reader is cautioned because this requires mentally reversing the order of the bits. Note, however, that in appendix B we demonstrate that no bit reversing is actually necessary in the practical implementation, which is just a linear feedback shift register in Galois configuration which is loaded with the input and stepped 64 times.

Likewise in the 32-bit case we define

$$
\begin{aligned}
p_{32}(x) \quad = \quad & x^{32} + x^{31} + x^{24} + x^{22} + x^{19} + x^{17} + \\
& x^{13} + x^{12} + x^{11} + x^{9} + x^{8} + x^{5} + x^{4} + x^{2} + 1
\end{aligned} \tag{3}
$$

(which is primitive) and let $C_{p_{32}}$ be the companion matrix of $p_{32}(x)$. For $v \in \mathbb{F}_2^{32}$ a row vector, we define

$$L_{32}(v) = vC_{p_{32}}^{32}.$$

Finally, to be complete with our definition we define

$$L(v) = \begin{cases} L_{64}(v) & \text{if } v \text{ is 64-bit} \\ L_{32}(v) & \text{if } v \text{ is 32-bit} \end{cases} \tag{4}$$

3.3.1 Selection Criteria for L

The first selection criteria for L (for both 32 and 64 bits) was that it should be a linear transformation whose rational canonical form (i.e. Fröbenious form) consists of a single irreducible block. This prevents the existence of any proper, non-trivial, invariant subspaces of the linear transformation. An easy method to construct such a matrix is to use the companion matrix, $C_{p(x)}$, of a primitive polynomial, $p(x)$. The minimal polynomial of such a matrix is $p(x)$. Hence the minimal polynomial is irreducible, so the linear transformation has no proper non-trivial invariant subspaces. (Likewise, it has no eigenvalues in the ground field.)

This matrix has order $2^n - 1$ in the group of invertible $n \times n$ matrices over $\mathbb{F}_2$. (In fact, the subgroup generated by the matrix is isomorphic to the multiplicative subgroup of the field $\mathbb{F}_2[x]/(p(x))$, and the action of $C_{p(x)}$ is equivalent to multiplication by $\bar{x}$ in $\mathbb{F}_2[x]/(p(x))$.)

After forming the companion matrix, $C_{p(x)}$ we take $C_{p(x)}^{\deg(p(x))}$ to form the matrix which represents the linear transformation. For our purposes, we are using polynomials of degree 32 or 64. Since any positive power of 2 is relatively prime to $2^n - 1$, the minimal polynomial of L is also $p(x)$, thus L also has no proper non-trivial invariant subspaces.

The second criteria for L is, informally, "every bit of input should affect approximately half of the output bits, and every bit of output should be affected by approximately half of the input bits." More precisely, given a vector, v, over $\mathbb{F}_2$, we define the *bias* of v as

$$\text{bias}(v) = \left| \frac{1}{2} - \frac{\text{number of ones in v}}{\text{number of elements in v}} \right|.$$

Given a matrix, M, over $\mathbb{F}_2$ let S_M be the set of all row and column vectors of M. We define the *matrix bias* of M as,

$$\text{bias}(M) = \max_{v \in S_M} \text{bias}(v).$$

The formal statement of the criteria is then: given a matrix M representing the linear transformation L with respect to the standard basis, we wish to minimize the bias of M.

To use a linear feedback shift register (LFSR) to implement the linear function and satisfy the second criteria, the register must be shifted at least as many times as it has entries. This is why the companion matrices $C_{p_{64}}$ and $C_{p_{32}}$ are raised to the 64^{th} and 32^{nd} powers respectively.

The search algorithm used to find p_{64} and p_{32} was to pseudo-randomly search for primitive polynomials, form the companion matrices of the polynomials, take the 64^{th} or 32^{nd} power of the companion matrix and compute its bias, keeping the polynomials which generated the smallest bias.

The biases for $C_{p_{64}}^{64}$ and $C_{p_{32}}^{32}$ are both 0.093750. Thus, for L_{64} at least 26 out of 64 bits of input affect each bit of output and likewise at least 26 out of 64 bits of output are affected by each bit of input. For L_{32} at least 13 out of 32

bits of input affect each bit of output and likewise at least 13 out of 32 bits of output are affected by each bit of input.

3.4 Avalanche Criteria for E

One important criterion is to determine the number of steps of the registers in E needed to ensure that every bit of input has, on average, a 50% chance of affecting any given bit of output. This was determined heuristically by Monte-Carlo methods to be 16 steps.

3.5 Differential Cryptanalysis of E

The first line of defense against differential cryptanalysis based attacks is that the F function has been chosen to minimize the differential characteristic of the shift registers when viewed as 8-bit functions.

Also, the entry in the differential table for the 7-bit function F with maximum probability has a probability of $\frac{82}{127 \cdot 64}$. We use this information to determine the upper bound on the number of times we need to step the system to resist a differential cryptanalysis attack on the F function. Equation (5) gives the probability that a given differential characteristic is propagated to a single bit through n steps.

$$\left(\frac{82}{127 \cdot 64}\right)^{an/8} \tag{5}$$

Where n is the number of steps, and a is the minimum number shift-registers which are involved in a single bit of output. (This is similar to the number of "active S-boxes" described in other cryptographic literature). The value of a comes from the bias of L as a is just the minimum number of inputs to L affecting a single bit of output. Hence $a = 26$ for L_{64} and $a = 13$ for L_{32}.

The number of steps is divided by 8 in the exponent because a given bit is modified only once every 8 steps.

In the 512-bit case we have

$$\left(\frac{82}{127 \cdot 64}\right)^{(26)n/8} < \frac{1}{2^{512}}$$

which yields

$$n > \frac{512 \cdot 8}{(26)(6 + \log_2(127) - \log_2(82))} > 23.76.$$

In the 256-bit case we have

$$\left(\frac{82}{127 \cdot 64}\right)^{(13)n/8} < \frac{1}{2^{256}}$$

which also yields

$$n > \frac{256 \cdot 8}{(13)(6 + \log_2(127) - \log_2(82))} > 23.76.$$

So, in both cases we take $n = 24$ to be the number of steps to defend against a differential cryptanalysis based attack.

3.6 Linear Cryptanalysis of E

The best affine approximation of the F function is the linear function

$$\ell(a, b, c, d, e, f, g) = a + d + g,$$

which agrees with F for 79 out of 128 possible inputs (or 15/128 more times than random guessing).

If we form a LFSR by replacing F with ℓ in Figure (3), then we wish to know, given any equal initial fill, how many steps of the register are required before the LFSR based on ℓ is different from the F based shift register. From an exhaustive search, we determined that the worst case occurs when the initial fill is all ones, and in this case it takes 12 steps before the registers diverge.

Therefore, suppose then that we form a permutation, E′, by replacing F with ℓ in the description of E. After 12 steps `r0` is now different in E and E′. After 13 steps, we expect at least 26/64 = 13/32 of the bit positions in `r0` to differ between E and E′ (due to our criterion for L). After 15 steps, we expect, on average, that half of the bits in `r0` will differ between E and E′. After 8 more steps, we expect that half of the bits of E are now different from E′.

Therefore we expect to need at least 23 steps to defend against a linear cryptanalysis based attack. So, we take 24 steps just as in the case for defending against differential cryptanalysis.

4 The ESSENCE Compression Function

The ESSENCE compression function, G, consists of two instances of the E permutation illustrated in Figure 4. One instance, E_K, shown by `k0-k7` in Figure 4, plays a role similar to key scheduling in a traditional block cipher system, and does not receive input while stepping. The other instance, E_R, shown by `r0-r7` in Figure 4, receives input from E_K. Figure 5 gives the pseudo-code for the compression function stepping logic.

To be precise, there are two compression functions. By G_{64} we mean the case where all of `r0-r7` and `k0-k7` are 64-bit integers. By G_{32} we mean the case where all of `r0-r7` and `k0-k7` are 32-bit integers. When the discussion applies to both, we shall simply refer to G.

To define G, let R and K each be blocks consisting of eight integers. We define G(R,K,n) as follows:

1. Initialize `r0` to R[0], `r1` to R[1], etc. and `k0` to K[0], `k1` to K[1], etc.

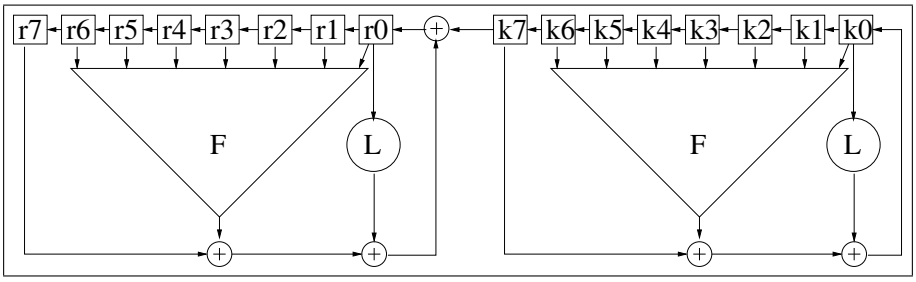

Figure 4: The ESSENCE Compression Function Logic

```
tmp_r := F(r6,r5,r4,r3,r2,r1,r0) xor r7 xor L(r0)
r7 := r6
r6 := r5
r5 := r4
r4 := r3
r3 := r2
r1 := r1
r1 := r0
r0 := tmp_r xor k7
tmp_k := F(k6,k5,k4,k3,k2,k1,k0) xor k7 xor L(k0)
k7 := k6
k6 := k5
k5 := k4
k4 := k3
k3 := k2
k2 := k1
k1 := k0
k0 := tmp_k
```

Figure 5: Pseudo-code for Compression Function Logic

2. Step the logic n times

3. The value of G(R,K,n) is defined to be the values in `r0-r7` bitwise xored with the values `R[0]-R[7]`.

The value of n, the number of times that the logic should be stepped, is implementation dependent. From the analysis of differential and linear cryptanalytic attacks given in sections 3.5 and 3.6, n should be at least 24 and should be a multiple of 8. As a measure of caution, we recommend that a value of $n = 32$ be used.

References

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

[BS91] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology—CRYPTO '90 Proceedings*, pages 2–21. Springer-Verlag, 1991.

[Gol67] Solomon W. Golomb. *Shift Register Sequences, Revised Edition*. Aegean Park Press (Orig. ed. Holden-Day), 1982 (Orig. ed. 1967).

[Sie84] T. Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographing applications. *IEEE Transactions on Information Theory*, IT-30(5):776–780, September 1984.

A Implementation Notes for F

The description given for F in equation (1) is not optimal for most compilers. An alternative expression for the F function for software implementation is

$$
\begin{aligned}
F(a,b,c,d,e,f,g) \quad = \quad & ab'cdefg' + abcef'g + abde'g + \\
& abefg + ab'def + acefg + ac'dfg' + \\
& adef'g + bcdfg + bdef'g + cdefg' + \\
& abc'g + a'bcf' + bce'g + bc'de + abf + \\
& ab'e + acg + adg' + aeg + a'ef + bdf' + \\
& beg + b'fg + cd'e' + c'df + def + \\
& de'g + dfg + a' + b + f
\end{aligned} \tag{6}
$$

where the symbol a' indicates the bit-wise negation of a, and the addition and multiplication are taken bit-wise in $\mathbb{F}_2$ (i.e. addition is just bit-wise XOR and multiplication is bit-wise AND.)

A standard C macro which will implement the F function is given in Figure 6.

For a hardware implementation, the F function can be implemented by two-level AND-OR (or NAND-NAND) logic using the prime implicants listed in Table 1. The prime implicants listed were obtained from the Quine-McCluskey algorithm.

```
#define F_func(a,b,c,d,e,f,g)                    \
  (                                              \
   ((a)&(~(b))&(c)&(d)&(e)&(f)&(~(g)))^          \
   ((a)&(b)&(c)&(e)&(~(f))&(g))^                 \
   ((a)&(b)&(d)&(~(e))&(g))^                     \
   ((a)&(b)&(e)&(f)&(g))^                        \
   ((a)&(~(b))&(d)&(e)&(f))^                     \
   ((a)&(c)&(e)&(f)&(g))^                        \
   ((a)&(~(c))&(d)&(f)&(~(g)))^                  \
   ((a)&(d)&(e)&(~(f))&(g))^                     \
   ((b)&(c)&(d)&(f)&(g))^                        \
   ((b)&(d)&(e)&(~(f))&(g))^                     \
   ((c)&(d)&(e)&(f)&(~(g)))^                     \
   ((a)&(b)&(~(c))&(g))^                         \
   ((~(a))&(b)&(c)&(~(f)))^                      \
   ((b)&(c)&(~(e))&(g))^                         \
   ((b)&(~(c))&(d)&(e))^                         \
   ((a)&(b)&(f))^                                \
   ((a)&(~(b))&(e))^                             \
   ((a)&(c)&(g))^                                \
   ((a)&(d)&(~(g)))^                             \
   ((a)&(e)&(g))^                                \
   ((~(a))&(e)&(f))^                             \
   ((b)&(d)&(~(f)))^                             \
   ((b)&(e)&(g))^                                \
   ((~(b))&(f)&(g))^                             \
   ((c)&(~(d))&(~(e)))^                          \
   ((~(c))&(d)&(f))^                             \
   ((d)&(e)&(f))^                                \
   ((d)&(~(e))&(g))^                             \
   ((d)&(f)&(g))^                                \
   ((~(a)))^                                     \
   ((b))^                                        \
   ((f))                                         \
                                                 \
                                                 )
```

Figure 6: F Implemented as a C Macro

Prime Implicants for F	
a'b'c'd'e'f'g'	a'bcde'fg'
a'b'c'd'e'f'g	a'bcde'fg
a'b'c'd'e'fg	ab'c'd'e'fg'
a'b'c'd'ef'g'	ab'c'd'ef'g'
a'b'c'd'ef'g	ab'c'de'f'g'
a'b'c'd'efg'	ab'c'de'f'g
a'b'c'de'f'g'	ab'c'de'fg
a'b'c'de'fg'	ab'c'def'g
a'b'c'def'g'	ab'c'defg'
a'b'c'def'g	ab'cd'e'f'g'
a'b'c'defg'	ab'cd'ef'g'
a'b'c'defg	ab'cd'ef'g
a'b'cd'e'fg'	ab'cde'f'g'
a'b'cd'ef'g'	ab'cde'fg
a'b'cd'ef'g	ab'cdefg'
a'b'cd'efg'	ab'cdefg
a'b'cde'f'g'	abc'd'e'f'g'
a'b'cde'fg	abc'd'e'fg'
a'b'cdef'g'	abc'd'ef'g'
a'b'cdef'g	abc'd'efg'
a'b'cdefg'	abc'd'efg
a'bc'd'e'fg'	abc'de'f'g'
a'bc'd'e'fg	abc'de'f'g
a'bc'd'ef'g	abc'defg
a'bc'd'efg	abcd'ef'g'
a'bc'de'f'g'	abcd'ef'g
a'bc'defg'	abcd'efg'
a'bc'defg	abcde'f'g'
a'bcd'e'f'g	abcde'fg
a'bcd'e'fg	abcdef'g'
a'bcd'ef'g'	abcdefg
a'bcd'efg	

Table 1: Prime Implicants for F

B Implementation Notes for L

The companion matrix of a polynomial acts on row vectors exactly as a linear feedback shift register in Galois configuration. Hence, it is extremely easy to implement multiplication by a companion matrix as a LFSR. However, efficient software implementation for a LFSR in Galois configuration is subtly due to the instruction pipeline used in modern processors. The naive approach, as demonstrated in Figure 7, is to use a conditional branch inside a loop. The problem with this approach is that the branch prediction logic in most processors cannot predetermine which branch will be taken. Since the loop is very small, speculative loading of both branches is also ineffective as the next iteration of the loop presents another unpredictable branch within a few machine instructions. Thus, the instruction decoding pipeline inside the processor must stall completely for each iteration of the stepping loop. On the Intel Core2 architecture this was measured to be a delay of 12-15 clock cycles for each iteration of the loop.

However, the conditional branch can be removed by taking advantage of signed integer arithmetic, provided the compiler and platform support a signed arithmetic right shift which copies the sign bit. Figure 8 demonstrates C code which, on all compilers the author tested for the x86 architecture, produces correct results. (The Gnu, Intel, and Microsoft compilers for the x86 family emit the `SAR` assembly instruction for arithmetic shift right. The `SAR` instruction shifts the bits of the integer to the right while duplicating the most significant bit. Thus performing an arithmetic shift right of 63 bits on a 64-bit signed integer has the effect of filling the entire result with the value of the most significant bit of the integer.)

A second advantage of this approach is that it allows for SIMD (Single Instruction Multiple Data) capable processors to execute multiple LFSR instances in parallel. For example, the 128-bit SSE instructions in the x86_64 family of processors can execute two instances of the L_{64} function or four instances of the L_{32} function in parallel on a single processor core.

A third advantage is that every step takes the same amount of time ensuring that no information can be leaked via timing side-channel attacks.

The implementer is strongly cautioned, however, that the ANSI C99 standard does not require this behavior. The C99 standard explicitly leaves the behavior of arithmetic shift right on signed integers as compiler dependent. A careful C implementation of the linear function should verify the behavior before using it.

A similar software implementation of L_{32} is given in Figure 9.

If the implementer is willing to use data-dependent look-up tables (which might make the implementation vulnerable to timing-attacks), the L function can be made considerably faster by pre-computing look-up tables values based upon the most significant byte of input. This trick is illustrated for L_{64} in Figure 10 and for L_{32} in Figure 11.

Note that in both Figures 10 and 11 there is an initialization routine that is used to build the look-up table. This routine need not be part of an implementations since the table can simply be pre-computed and included in the code.

```
/*
 * 64-bit integer representing the p_64 polynomial
 */
#define P_64 0xb0a65313e6966997LL


/*
 * An inefficient implementation of L_64
 */
uint64_t L_64(uint64_t input)
  {
    int i;
    uint64_t temp;

    temp = input;
    for (i=0;i<64;i++)
      {
        /*
         * SLOW:  Un-predictable conditional branch
         * stalls the instruction pipeline.  This might
         * even create a timing attack problem.  Do not
         * use this method... unless you're running on
         * an Itanium or similar processor which can
         * use speculative executio and predication to
         * execute both branches in parallel.
         */
        if ((temp & 0x8000000000000000LL) ==
            0x8000000000000000LL)
          {
            temp = (temp << 1) & P_64;
          }
        else
          {
            temp <<= 1;
          }
      }
    return(temp);
  }
```

Figure 7: Inefficient L_{64} Implementation in C

```
/*
 * 64-bit integer representing the p_64 polynomial
 */
#define P_64 0xb0a65313e6966997LL


/*
 * An efficient implementation of L_64
 */
uint64_t L_64(uint64_t input)
  {
    int i;
    int64_t temp;

    temp = (int64_t)input;  /* Convert to signed integer */
    for (i=0;i<64;i++)
      {
        /*
         * CAUTION:  Arithmetic Shift Right of signed integers
         * is compiler dependent.  It is left unspecified by
         * the ANSI standard.  The implementor must verify
         * that the sign bit (most significant bit) is
         * preserved.
         */
        temp = ((temp >> 63) & P_64) ^ (temp << 1);
      }
    return((uint64_t)temp);
  }
```

Figure 8: Efficient L_{64} Implementation in C

```
/*
 * 32-bit integer representing the p_32 polynomial
 */
#define P_32 0x814a3b35


/*
 * An efficient implementation of L_32
 */
uint32_t L_32(uint32_t input)
  {
    int i;
    int32_t temp;

    temp = (int32_t)input;  /* Convert to signed integer */
    for (i=0;i<32;i++)
      {
        /*
         * CAUTION:  Arithmetic Shift Right of signed integers
         * is compiler dependent.  It is left unspecified by
         * the ANSI standard.  The implementor must verify
         * that the sign bit (most significant bit) is
         * preserved.
         */
        temp = ((temp >> 31) & P_32) ^ (temp << 1);
      }
    return((uint32_t)temp);
  }
```

Figure 9: Efficient L_{32} Implementation in C

We give the routines here just to clarify how such tables would be computed (and because listing the tables would be tedious).

To see how this method works in the case of L_{64} consider the row vector $v \in \mathbb{F}_2^{64}$. Write $v = [v_0, v_1, \ldots, v_{62}, v_{63}]$, and then let $u = [v_0, v_1, \ldots, v_{55}, 0, 0, \ldots, 0]$ and $w = [0, 0, \ldots, 0, v_{56}, v_{57}, \ldots, v_{63}]$. So $v = u + w$. Now recall that $C_{p_{64}}$ has ones on the super-diagonal, the coefficients from p_{64} along the bottom row and zeros elsewhere. So, the matrix structure of $C_{p_{64}}^8$ has ones along the ninth-diagonal non-zero values in the last eight rows, and zero everywhere else:

$$C_{p_{64}}^8 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \ldots & 0 \\ & & & & & & \vdots & & & & \ddots & & \\ & & & & & & \vdots & & & & & \ddots & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 1 \\ * & * & * & * & * & * & * & * & * & * & * & \ldots & * \\ & & & & & & \vdots & & & & & & \\ * & * & * & * & * & * & * & * & * & * & * & \ldots & * \end{pmatrix}.$$

Hence $uC_{p_{64}}^8$ is just equal to u shifted by eight positions since u is zero in the last eight positions. For $wC_{p_{64}}^8$ we consult a look-up table which is easily pre-computed because w can only take on 2^8 values. Since $vC_{p_{64}}^8 = (u + w)C_{p_{64}}^8 = uC_{p_{64}}^8 + wC_{p_{64}}^8$, we have replaced 8 matrix multiplications with a shift and a table look-up. We repeat this 8 times to compute L_{64}. A similar observation works for L_{32}. One advantage of this approach is that the processor only needs to deal with 8-bit shifts. On an 8-bit processor, we may thus avoid having to shift bits across byte boundaries and simply move the bytes.

While a direct hardware implementation of the LFSR is simple, it requires a counter and 64 or 32 clock cycles. Therefore, hardware implementers are advised to simply construct the matrices $C_{p_{64}}^{64}$ and $C_{p_{32}}^{32}$ as the linear logic is very simple and can execute in one clock cycle.

```
/*
 * This FAST look-up table based implementation of L_64
 * might be vulnerable to timing attacks.
 */
void init_L_64_table(uint64_t L_64_table[])
  {
    int i,j;
    uint64_t lin_tmp;

    for(i=0;i<256;i++)
      {
        lin_tmp = ((uint64_t)i) << 56;
        for(j=0;j<8;j++)
          {
            lin_tmp = ((( ((int64_t)lin_tmp) >> 63) & P_64) ^
                       (lin_tmp << 1));
          }
        L_64_table[i] = lin_tmp;
      }
  }

uint64_t L_64(uint64_t input, uint64_t L_64_table[])
  {
    int i;
    for (i=0;i<8;i++)
      {
        input = L_64_table[(input >> 56) & 0xff] ^
                (input << 8);
      }
    return(input);
  }
```

Figure 10: Fast Look-up Table L_{64} Implementation in C

```
/*
 * This FAST look-up table based implementation of L_32
 * might be vulnerable to timing attacks.
 *
 */
void init_L_32_table(uint32_t L_32_table[])
  {
    int i,j;
    uint32_t lin_tmp;

    for(i=0;i<256;i++)
      {
        lin_tmp = ((uint32_t)i) << 24;
        for(j=0;j<8;j++)
          {
            lin_tmp = (  ( ((int32_t)lin_tmp) >> 31 ) & P_32  ) ^
                      (lin_tmp << 1);
          }
        L_32_table[i] = lin_tmp;
      }
  }

uint32_t L_32(uint32_t input, uint32_t L_32_table[])
  {
    int i;
    for (i=0;i<4;i++)
      {
        input = L_32_table[(input >> 24) & 0xff] ^
                (input << 8);
      }
    return(input);
  }
```

Figure 11: Fast Look-up Table L_{32} Implementation in C

C Implementation Notes for the Compression Function

On most 64-bit processors (and many 32-bit processors) the two instances of E in the compression function can be implemented in parallel at the instruction level by taking advantage of the largest registers available in the processor. For example, to implement G_{32} on a 64-bit processor, we place `r0` and `k0` in the same 64-bit integer: `r0` occupies the most significant 32-bits and `k0` occupies the least significant 32-bits. Likewise, we place the other `r` and `k` values in the upper and lower portions of 64-bit integers. Then, a single evaluation of F on the 64-bit integers simultaneously computes `F(r6,r5,r4,r3,r2,r1,r0)` and `F(k6,k5,k4,k3,k2,k1,k0)`. On machines supporting 128-bit SIMD registers and instructions (such as the SSE instructions on the x86_64 family of processors), one may likewise compute the two instances of E in G_{64} in parallel.

D Explicit Values of the F function

The following tables give the explicit value for the F function for each possible input. The 128 possible values for F have been divided up into four tables of 32 values each based upon the value of the first two entries.

a	b	c	d	e	f	g	F(a,b,c,d,e,f,g)
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0
0	0	0	0	0	1	1	1
0	0	0	0	1	0	0	1
0	0	0	0	1	0	1	1
0	0	0	0	1	1	0	1
0	0	0	0	1	1	1	0
0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	0	1	0	1
0	0	0	1	0	1	1	0
0	0	0	1	1	0	0	1
0	0	0	1	1	0	1	1
0	0	0	1	1	1	0	1
0	0	0	1	1	1	1	1
0	0	1	0	0	0	0	0
0	0	1	0	0	0	1	0
0	0	1	0	0	1	0	1
0	0	1	0	0	1	1	0
0	0	1	0	1	0	0	1
0	0	1	0	1	0	1	1
0	0	1	0	1	1	0	1
0	0	1	0	1	1	1	0
0	0	1	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	1	0	1	0	0
0	0	1	1	0	1	1	1
0	0	1	1	1	0	0	1
0	0	1	1	1	0	1	1
0	0	1	1	1	1	0	1
0	0	1	1	1	1	1	0

Table 2: Values of F with (a,b) = (0,0)

a	b	c	d	e	f	g	F(a,b,c,d,e,f,g)
0	1	0	0	0	0	0	0
0	1	0	0	0	0	1	0
0	1	0	0	0	1	0	1
0	1	0	0	0	1	1	1
0	1	0	0	1	0	0	0
0	1	0	0	1	0	1	1
0	1	0	0	1	1	0	0
0	1	0	0	1	1	1	1
0	1	0	1	0	0	0	1
0	1	0	1	0	0	1	0
0	1	0	1	0	1	0	0
0	1	0	1	0	1	1	0
0	1	0	1	1	0	0	0
0	1	0	1	1	0	1	0
0	1	0	1	1	1	0	1
0	1	0	1	1	1	1	1
0	1	1	0	0	0	0	0
0	1	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	0	0	1	1	1
0	1	1	0	1	0	0	1
0	1	1	0	1	0	1	0
0	1	1	0	1	1	0	0
0	1	1	0	1	1	1	1
0	1	1	1	0	0	0	0
0	1	1	1	0	0	1	0
0	1	1	1	0	1	0	1
0	1	1	1	0	1	1	1
0	1	1	1	1	0	0	0
0	1	1	1	1	0	1	0
0	1	1	1	1	1	0	0
0	1	1	1	1	1	1	0

Table 3: Values of F with (a,b) = (0,1)

a	b	c	d	e	f	g	F(a,b,c,d,e,f,g)
1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0
1	0	0	0	0	1	0	1
1	0	0	0	0	1	1	0
1	0	0	0	1	0	0	1
1	0	0	0	1	0	1	0
1	0	0	0	1	1	0	0
1	0	0	0	1	1	1	0
1	0	0	1	0	0	0	1
1	0	0	1	0	0	1	1
1	0	0	1	0	1	0	0
1	0	0	1	0	1	1	1
1	0	0	1	1	0	0	0
1	0	0	1	1	0	1	1
1	0	0	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	1	0	0	0	0	1
1	0	1	0	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	0	0	1	1	0
1	0	1	0	1	0	0	1
1	0	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	0	1	1	1	0
1	0	1	1	0	0	0	1
1	0	1	1	0	0	1	0
1	0	1	1	0	1	0	0
1	0	1	1	0	1	1	1
1	0	1	1	1	0	0	0
1	0	1	1	1	0	1	0
1	0	1	1	1	1	0	1
1	0	1	1	1	1	1	1

Table 4: Values of F with (a,b) = (1,0)

a	b	c	d	e	f	g	F(a,b,c,d,e,f,g)
1	1	0	0	0	0	0	1
1	1	0	0	0	0	1	0
1	1	0	0	0	1	0	1
1	1	0	0	0	1	1	0
1	1	0	0	1	0	0	1
1	1	0	0	1	0	1	0
1	1	0	0	1	1	0	1
1	1	0	0	1	1	1	1
1	1	0	1	0	0	0	1
1	1	0	1	0	0	1	1
1	1	0	1	0	1	0	0
1	1	0	1	0	1	1	0
1	1	0	1	1	0	0	0
1	1	0	1	1	0	1	0
1	1	0	1	1	1	0	0
1	1	0	1	1	1	1	1
1	1	1	0	0	0	0	0
1	1	1	0	0	0	1	0
1	1	1	0	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	0	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	0	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	0	0	0	1
1	1	1	1	0	0	1	0
1	1	1	1	0	1	0	0
1	1	1	1	0	1	1	1
1	1	1	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1

Table 5: Values of F with (a,b) = (1,1)