Mercy: a fast large block cipher for disk sector encryption

Paul Crowley
mercy@paul.cluefactory.org.uk

DataCash Ltd.

Abstract. We discuss the special requirements imposed on the underlying cipher of systems which encrypt each sector of a disk partition independently, and demonstrate a certificational weakness in some existing block ciphers including Bellare and Rogaway's 1999 proposal, proposing a new quantitative measure of avalanche. To address these needs, we present Mercy, a new block cipher accepting large (4096-bit) blocks, which uses a key-dependent state machine to build a bijective F function for a Feistel cipher. Mercy achieves 9 cycles/byte on a Pentium compatible processor.
Key words: disk sector, large block, state machine, avalanche, Feistel cipher.
Mercy home page: *http://www.cluefactory.org.uk/paul/mercy/*

1 Introduction

Disk sector encryption is an attractive approach to filesystem confidentiality. Filesystems access hard drive partitions at the granularity of the sector (or block) where a sector is typically 4096 bits: read and write requests are expressed in sector numbers, and data is read and modified a sector at at a time. Disk sector encryption systems present a "virtual partition" to the filesystem, mapping each sector of the virtual partition to the corresponding sector, through an encrypting transformation, on a physical disk partition with the same disk geometry. The performance is typically better than file-level encryption schemes, since every logical sector read or write results in exactly one physical sector read or write, and confidentiality is also better: not only are file contents obscured, but also filenames, file sizes, directory structure and modification dates. These schemes are also flexible since they make no special assumptions about the way the filesystem stores the file data; they work equally well with raw database partitions as with filesystems, and can be transparently layered underneath disk caching and disk compression schemes. Linux provides some support for such filesystems through the "/dev/loop0" filesystem device.

The stream cipher SEAL [17] is well suited to this need. SEAL provides a strong cryptographic PRNG (CPRNG) whose output is seekable. Thus the entire disk can be treated as a single contiguous array of bytes and XORred with the output from the CPRNG; when making reads or writes of specific sectors the appropriate portion of the output can be generated without the need to generate the preceding bytes. The same effect can be achieved, somewhat less efficiently, by keying a CPRNG such as ARCFOUR [10] with a (key, sector number) pair and generating 512 bytes with which to encrypt the sector. These schemes are highly efficient and provide good security against an attacker who seizes an encrypted hard drive and attempts to gain information about its contents.

However, this is not strong against attackers with other channels open to them. They may have user privileges on the system they're trying to attack, and be able to access the ciphertext stored on the hard drive at times when it's shut down. Or they may try to modify sectors with known contents while carrying a drive from place to place. They may even be able to place hardware probes on the drive chain while logged in as a normal user, and sniff or modify ciphertext. Against these attacks, SEAL and ARCFOUR (used as described) are ineffective. For example, an attacker can write a large file of all zeroes and thereby find the fixed encryption stream associated with many sectors; once the file is deleted, the sectors might be re-used by other users with secure data to write, and this data is easily decrypted by XORing with the known stream. Or, if attackers can make a guess of the plaintext in a given sector, they can modify this to another plaintext of their choosing while they have access to the drive by XORing the ciphertext with the XOR difference between the two plaintexts.

File-based encryption schemes defeat these attacks by using a new random IV for each new plaintext and authenticating with a MAC. However, applying these techniques directly to sector encryption would require that the ciphertext for each sector be larger than the plaintext, typically by at least 64 bytes. Thus either the plaintext sectors would need to be slightly smaller than the natural hardware sector size, harming performance when mapping files into memory (and necessitating a thorough re-engineering of the filesystem code) or auxiliary information would have to be stored in other sectors, potentially adding a seek to each read and write. In either case the size overhead will be about 1.5 - 3.1%. It's worth investigating what can be achieved without incurring these penalties.

SFS [9] uses a keyless mixing transformation on the plaintext before applying a block chaining stream cipher. This greatly reduces the practical usefulness of many such attacks, but it falls short of the highest security that pure disk sector encryption systems can aspire to: that the mapping between each virtual and physical disk sector appears to be an independent random permutation to an attacker who expends insufficient computation to exhaustively search the keyspace. In other words, the theoretical best solution under these constraints is a strong randomised large block cipher.

Several proposals exist for building large block ciphers from standard cryptographic components such as hash functions and stream ciphers [1, 11, 2]; however, these are not randomised ciphers, and as Section 2 shows, they have certificational weaknesses. More seriously, no proposal comes close to offering the performance needed: bit rates equal to or better than disk transfer rates. Since small improvements in disk access efficiency can mean big improvements to every part of the user experience, and since performance considerations are one of the main reasons why filesystem encryption is not more widely used, it seems worthwhile to develop a new cipher designed specifically to meet this need.

The rest of this paper is organised as follows. Section 2 describes a certificational weakness applicable to several existing classes of large block cipher, and proposes a quantitiative measure of avalanche based on the attack. Section 3 lays out the design goals for the solution here, a new block cipher Mercy with a native block size of 4096 bits, and Section 4 describes the cipher itself. Section 5 discusses the design of the cipher in detail with reference to the performance and security goals of Section 3. Finally Section 6 discusses some of the lessons learned in the design of Mercy.

2 Avalanche and certificational weaknesses

[14] presents an attack on bidirectional MACs based on inducing collisions in the internal state of the MAC. This attack can be extended to show a certificational weakness in some large block cipher proposals. Note that neither keys nor plaintext can be recovered using this attack; it merely serves to distinguish the cipher from a random permutation.

In general form, the attack proceeds as follows. The bits in the plaintext are divided into two categories, "fixed" and "changing"; a selection of the bits of the ciphertext are chosen as "target" bits. A number of chosen plaintexts are encrypted, all with the same fixed bits and with changing bits chosen at random; the attack is a success if a collision in the target bits of the ciphertext is generated. Let w_k be the length of the key, w_t the number of target bits, and 2^{w_p} be the number of different plaintexts encrypted: if the following conditions are met:

- the result is statistically significant, ie the probability of seeing such a collision under these circumstances from a genuine PRP (approximately $2^{2w_p - w_t - 1}$) is small; and
- $w_p < w_k - 1$, ie the work factor for the attack is less than that for a key guessing attack against the cipher

then a certificational weakness has been demonstrated. The attack works by inducing an internal collision in the data path from the changing bits to the fixed bits; the width of this path determines the number of plaintexts needed and thus the smallest w_p for which the attack can work provides a useful quantitative measure of avalanche. This attack can easily be converted to use the memory-efficient parallel collision finding techniques of [20], so memory usage does not present a serious obstacle to the practicality of the attack if 2^{w_p} adaptive chosen plaintexts can be encrypted.

This attack may be applied to [2] by choosing the first two blocks of the plaintext as the "changing" bits, and all of the output except the second two blocks as the "target" bits. If the blocksize of the underlying cipher is 64 bits, then 2^{33} chosen plaintexts should be sufficient to induce a collision in σ, resulting in a collision in all the target bits as desired; if it is 128 bits, 2^{65} will be needed.

This attack also extends to bidirectional chaining systems, in which the plaintext is encrypted first forwards and then backwards using a standard block cipher in a standard chaining mode; in this case, the first two blocks are the changing bits, all of the ciphertext except the first two blocks are the target bits, and the number of plaintexts required are as before; the collision is induced in the chaining state after the first two blocks. If the chaining mode is CBC or CFB, all of the output except the first block will be target bits, since a collision in the internal state after the second block means that the second block of ciphertext is identical.

Note that this attack is not applicable to any of the proposals in [1]; neither BEAR nor LION claim to be resistant to any kind of chosen plaintext attack, while LIONESS carries 256 or 320 bits of data between the two halves (depending on the underlying hash function), which would require 2^{129} or 2^{161} chosen plaintexts; this is outside the security goals of the cipher. However, it can be applied to BEAST from [11] by inducing a collision in the SHA-1 hash of R^{**} with 2^{81} chosen plaintexts; the changing bits are the first 160 bits of R^{**}, and the target bits are all of the ciphertext except the first 160 bits of T^{**}. This attack is clearly impractical at the moment but it violates our expectation that the cheapest way to distinguish a block cipher from a random permutation should be a brute force key guessing attack.

3 Mercy design goals

Mercy is a new randomised block cipher accepting a 4096-bit block, designed specifically for the needs of disk sector encryption; it achieves significantly higher performance than any large block cipher built using another cipher as a primitive, or indeed than any block cipher that I know of large or small.

It accepts a 128-bit randomiser; it is expected that the sector number will be used directly for this purpose, and therefore that most of the randomiser bits will usually be zero. This is also known as a "diversification parameter" in the terminology of [6], or "spice" in that of [19]. This last term avoids the misleading suggestion that this parameter might be random and is convenient for constructions such as "spice scheduling" and "spice material" and is used henceforth.

Mercy's keyschedule is based on a CPRNG; the sample implementation uses [10]. Though [10] takes a variable length key, Mercy does not aspire to better security than a cipher with a fixed 128-bit key size, so it's convenient for the purposes of specifying these goals to assume that the key is always exactly 128 bits.

- **Security**: Any procedure for distinguishing Mercy encryption from a sequence of 2^{128} independent random permutations (for the 2^{128} possible spices) should show no more bias towards correctness than a key guessing attack with the same work factor. However, we do not claim that ignorance of the spice used would make any attack harder; it's not intended that the spice be hidden from attackers. For this reason, Mercy is not intended to be a K-secure or hermetic cipher in the terminology of [7].

- **Resistance to specific attacks**: Mercy is designed to be resistant in particular to linear and differential attacks, as well as to avoid the certificational weakness of Section 2.
- **Speed**: Encryption and decryption should be much faster than disk transfer rates, even with fast disks and slow processors. Specifically, they should be faster than 20 Mbytes/sec on a relatively modest modern machine such as the author's Cyrix 6x86MX/266 (which has a clock frequency of 233 MHz). This translates as under 11.7 cycles/byte, within the range of stream ciphers but well outside even the fastest traditional block cipher rates. The current C implementation of Mercy achieves 9 cycles/byte; it is likely that an assembly implementation would do rather better.
- **Memory**: The cipher should refer to as little memory as possible, and certainly less than 4kbytes. In many environments, Mercy's keytables will be stored in unswappable kernel memory; more important however is to minimise the amount of Level 1 cache that will be cleared when the cipher is used. 1536 bytes of storage are used.
- **Simplicity**: Mercy is designed to be simple to implement and to analyse.
- **Decryption**: Decryption will be much more frequent than encryption and should be favoured where there is a choice.

4 Description of Mercy

Since most of Mercy's operations are based around 32-bit words, we define $Z_w = Z_{2^{32}}$. Vectors are indexed from zero, so a vector $P \in Z_w^{128}$ of 128 32-bit numbers will be indexed as $(P_0, P_1, \ldots, P_{127})$. The symbol $\oplus$ represents bitwise exclusive OR; where + appears in the figures with a square box around it, it represents addition in the ring Z_w. Least significant and lowest indexed bytes and words appear leftmost and uppermost in the figures.

Note that some details that would be needed to build a specification of Mercy-based file encryption sufficient for interoperability, such as byte ordering within words, are omitted since they are irrelevant for cryptanalytic purposes.

4.1 T box

The T box ($T : Z_{256} \to Z_w$, Figure 1) is a key-dependent mapping of bytes onto words. N represents multiplicative inverses in $GF(2^8)$ with polynomial base $x^8 + x^4 + x^3 + x + 1$ except that $N(0) = 0$. $d_0 \ldots d_7$ are key dependent bijective affine mappings on $GF(2)^8$.

$$T(x) = \sum_{i=0}^{3} 2^{8i} d_{2i+1}[N(d_{2i}[x])]$$

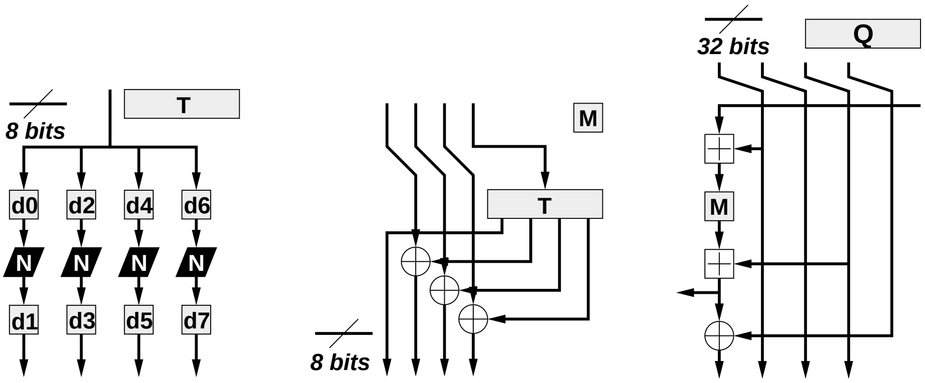

Fig. 1. T box, Operation M, Q state machine

4.2 Operation M

$M : Z_w \rightarrow Z_w$ (Figure 1) is drawn from David Wheeler's stream cipher WAKE [21]; it's a simple, key-dependent mapping on 32-bit words. The most significant byte of the input word is looked up in the T box, and the output XORred with the other three bytes shifted up eight bits; the construction of the T box ensures that this mapping is bijective.

$$M(x) = 2^8 x \oplus T\left(\lfloor x/2^{24} \rfloor\right)$$

4.3 Q state machine

The Q state machine (Figure 1) maps a four-word initial state and one word input onto a four-word final state and one word output ($Q : Z_w^4 \times Z_w \rightarrow Z_w^4 \times Z_w$) using taps from a linear feedback shift register and a nonlinear mixing function.

$$Q(S, x) = ((S_3 \oplus y, S_0, S_1, S_2), y) \quad \text{where} \quad y = S_2 + M(S_0 + x) \qquad (1)$$

4.4 F function

The F_n function ($n \geq 8$; Figure 2) accepts a 128-bit spice $G \in Z_w^4$ and a $32n$-bit input $A \in Z_w^n$ and generates a $32n$-bit output $B \in Z_w^n$ ($F_n : Z_w^n \times Z_w^4 \rightarrow Z_w^n$). F_{64} (usually just F) is the F function for the Feistel rounds. Here $S_{0 \ldots n+4}$ represents successive 128-bit states of a state machine; $U_{0 \ldots n+3} \in Z_w$ are the successive 32-bit inputs to the state machine, and $V_{0 \ldots n+3} \in Z_w$ are the outputs.

$$\begin{aligned} F_n(A,G) &= B \quad \text{where} \\ S_0 &= (A_{n-4}, A_{n-3}, A_{n-2}, A_{n-1}) \\ (S_{i+1}, V_i) &= Q(S_i, U_i) \qquad (0 \le i < n+4) \\ U_i &= \begin{cases} G_i & (0 \le i < 4) \\ A_{i+n-12} & (4 \le i < 8) \\ A_{i-8} & (8 \le i < n) \end{cases} \\ B_i &= \begin{cases} V_{i+8} & (i < n-4) \\ S_{n+4,i+4-n} & (n-4 \le i < n) \end{cases} \end{aligned}$$

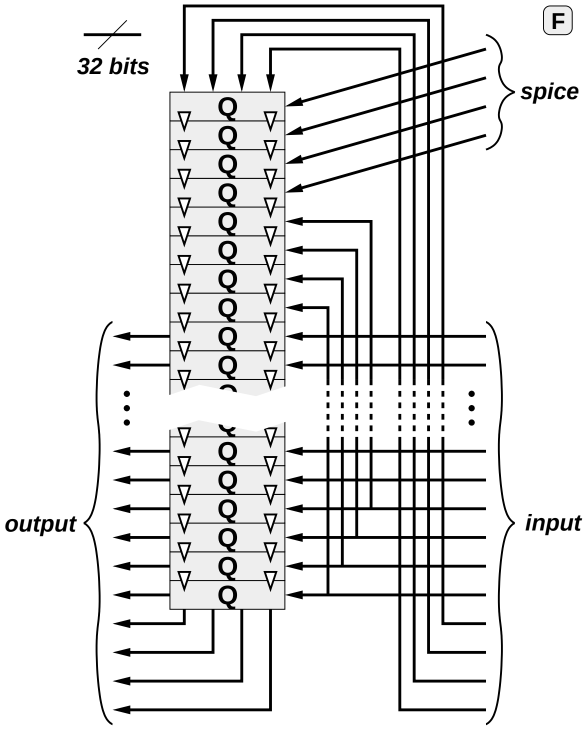

Fig. 2. F function

4.5 Round structure

Mercy uses a six round Feistel structure (Figure 3) with partial pre- and post-whitening; unusually, the final swap is *not* omitted. The spice $G \in Z_w^4$ (usually the sector number) goes through a "spice scheduling" procedure, analogous with

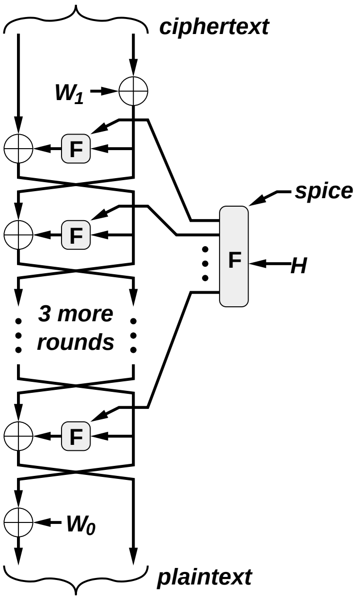

Fig. 3. Round structure (decryption illustrated)

key scheduling, through which the "spice material" $G' \in Z_w^{24}$ is generated based on the input spice, using the F_{24} variant of the F function; this forms six 128-bit "round spices". Spice scheduling uses a dummy input to the F function; for this a vector of incrementing bytes $H \in Z_w^{24}$ is used. $P \in Z_w^{128}$ represents the plaintext, $C \in Z_w^{128}$ the ciphertext, and $W_0, W_1 \in Z_w^{64}$ the whitening values. We describe decryption below; since Mercy is a straightforward Feistel cipher encryption follows in the straightforward way.

$$\begin{aligned} H_i &= \sum_{j=0}^{3} 2^{8j}(4i+j) \\ G' &= F_{24}(H, G) \\ (L_0, R_0) &= (C_{0\ldots63}, C_{64\ldots127} \oplus W_1) \\ (L_{i+1}, R_{i+1}) &= (R_i, L_i \oplus F_{64}(R_i, G'_{4i\ldots4i+3})) \qquad (0 \le i < 6) \\ (P_{0\ldots63}, P_{64\ldots127}) &= (L_6 \oplus W_0, R_6) \end{aligned}$$

4.6 Key schedule

The key is used to seed a CPRNG from which key material is drawn; [10] is used in the sample implementation (after discarding 256 bytes of output), and is convenient since it's small and byte oriented, but any strong CPRNG will serve. Then the procedure in Figure 4 generates the substitutions $d_{0\ldots7}$ and the 2048-bit whitening values W_0, W_1.

```
for $i \leftarrow 0 \ldots 7$ do
    $d_i[0] \leftarrow$ random_byte()
    for $j \leftarrow 0 \ldots 7$ do
        do
            $r \leftarrow$ random_byte()
        while $r \in d_i[0 \ldots 2^j - 1]$
        for $k \leftarrow 0 \ldots 2^j - 1$ do
            $d_i[k + 2^j] \leftarrow d_i[k] \oplus r \oplus d_i[0]$
for $i \leftarrow 0 \ldots 1$ do
    for $j \leftarrow 0 \ldots 64$ do
        $W_{i,j} \leftarrow 0$
        for $k \leftarrow 0 \ldots 3$ do
            $W_{i,j} \leftarrow W_{i,j} + 2^{8k}$random_byte()
```

Fig. 4. Key schedule pseudocode

An expected 10.6 random bytes will be drawn for each d. Once $d_{0 \ldots 7}$ have been determined, a 1k table representing the T box can be generated. During normal use 1536 bytes of key-dependent tables are used.

5 Design of Mercy

Existing approaches to large block ciphers use a few strong rounds based on existing cryptographic primitives. These ciphers cannot achieve speeds better than half that of the fastest block ciphers [2] or a third of the fastest stream ciphers [1]. Current block cipher speeds don't approach the needs of the design goals even before the extra penalties for doubling up, while those solutions based on stream ciphers pay a heavy penalty in key scheduling overhead that puts their speeds well below those needed.

Mercy addresses this by using more rounds of a weaker function. This makes more efficient use of the work done in earlier rounds to introduce confusion in later rounds, and is closer to a traditional block cipher approach drawn across a larger block. It also draws more state between disparate parts of the block, protecting against the certificational weaknesses identified in Section 2.

5.1 Balanced Feistel network

Balanced Feistel networks are certainly the best studied frameworks from which to build a block cipher, although I know of no prior work applying them to such large block sizes. They allow the design of the non-linear transformations to disregard efficiency of reversal and provide a familiar framework by which to analyse mixing. Balanced networks seem better suited to large block ciphers than unbalanced networks, since an unbalanced network is likely to have to do work proportional to the larger of the input and output data width.

Feistel ciphers normally omit the final swap, so that decryption has the same structure as encryption. However, Mercy implementations will typically encrypt blocks in-place, and the cost of having an odd number of swaps (forcing a real swap) would be high, so the last swap is not omitted.

Mercy's round function, while weaker than those used in [1], is considerably stronger than that of traditional Feistel ciphers, necessitating many fewer rounds. The larger block size allows more absolute work to be done in each round, while keeping the work per byte small.

5.2 Key schedule and S-boxes $d_{0\ldots 7}$

The function N used in building the T box is that used for nonlinear subsitution in [7]; it is bijective and has known good properties against linear and differential cryptanalysis, such as low LCmax and DCmax in the terminology of [13]. We use this function to build known good key-dependent bijective substitutions using an extension of the technique outlined in [4]; however, rather than a simple XOR, the d mappings before and after N are drawn at random from the entire space of bijective affine functions on $GF(2)^8$, of which there are approximately $2^{70.2}$, by determining first the constant term $d[0]$ and then drawing candidate basis values from the CPRNG to find a linearly independent set. Because the d functions are affine, the LCmax and DCmax of the composite function $d_1 \circ N \circ d_0$ (and siblings) will be the same as those of N itself. The composite functions will also be bijective since each of the components are, and hence all the bytes in each column of the T table will be distinct.

However, there are fewer possible composite functions than there are pairs d_0, d_1. In fact for each possible composite function, there are $255 \times 8 = 2040$ pairs d_0, d_1 which generate it. This follows from the following two properties of N:

$$\forall a \in GF(2^8) - \{0\} : \forall x \in GF(2^8) : aN(ax) = N(x)$$
$$\forall b \in 0 \ldots 7 : \forall x \in GF(2^8) : N(x^{2^j}) = N(x)^{2^j}$$

Since both multiplication and squaring in $GF(2^8)$ are linear (and hence affine) in $GF(2)^8$ (squaring because in a field of characteristic 2, $(x+y)^2 = x^2 + xy + yx + y^2 = x^2 + y^2$), both of these properties provide independent ways of mapping from any pair d_0, d_1 to pairs which will generate the same composite function, distinct in every case except $(a, b) = (1, 0)$. Taking this into account, the number of distinct composite functions possible is approximately $2^{129.4}$, and there are $2^{4613.7}$ functionally distinct keys in total (considering W_0, W_1 as well as T).

Little attention has been paid to either the time or memory requirements of the key schedule, since key scheduling will be very infrequent and typically carried out in user space.

5.3 Operation M

As with [21], this operation is bijective, since from the least significant byte of the output the input to the T box can be inferred. We differ from [21] in using the most significant byte for the lookup rather than the least; this slightly improves the mixing gained from addition and seems to be very slightly faster on the author's machine.

5.4 Q state machine

A key-dependent state machine is an efficient way to bring about strong dependencies between widely separated parts of the block; since the state machine is reversible, changing a given input word guarantees that every subsequent state will be different and makes it slightly more likely than chance that every output word will be different.

The basis of the state machine is 32 parallel four-stage LFSRs based on the polynomial x^4+x^3+1. The input to each LFSR is provided by a nonlinear mixing chain based on carry bits from other LFSRs and taps into the state which are then fed into Operation M after addition-based mixing with the input. The use of an LFSR ensures that any pair of distinct inputs of the same length which leave the LFSR in the same state must be at least 5 words long.

Every T box lookup depends on the previous lookup, even across rounds. This goes against the design principles outlined in [18, 5] which suggest that ciphers should be designed to make best use of the parallelism that modern processors can achieve, and to be wary of the memory latency of table lookups. A variant on Q which allows several T box lookups to take place in parallel by taking taps from later in the LFSR is easy to construct, but surprisingly did not result in any speed improvements on the target platform. Ciphers similar to Mercy which use this technique to improve parallelism may be appropriate for other architectures.

Since the Feistel rounds use XOR mixing, Q is also designed such that the first operation on the input is addition, as is the last operation on the output. This improves operation mixing, helping to frustrate attacks which model the cipher using a single algebraic group. XOR is also used within Q for the same reason.

The output is chosen for the property that, where $Q(S, x) = (S', y)$, if either of S or S' is known and either of x or y is known, the two unknowns can be inferred. We prove this in four cases below:

1. S, x known, S', y unknown: use Q directly.
2. S', x known, S, y unknown: from Equation 1 we infer $y = S'_3 + M(S'_1 + x)$ and $S = (S'_1, S'_2, S'_3, S'_0 \oplus y)$.
3. S, y known, S', x unknown: $x = M^{-1}(y - S_2) - S_0$ (defined since M is bijective), then apply Q as normal.
4. S', y known, S, x unknown: find S as in 2 and x as in 3.

5.5 F function

These properties of Q are used to build a fast, bijective F function. A bijective F function lends resistance to a number of attacks, making 2 round iterative differential characteristics impossible and frustrating truncated differential cryptanalysis, as well as lending confidence that the input is being used efficiently. For a fixed spice G, we infer F's input A given the output B as follows: the final state of the state machine is simply the last four words of the output: $S_{n+4} = B_{n-4\ldots n-1}$; from this, we can run the state machine backwards up to S_8 as with point 4 of Section 5.4 above, inferring $A_{0\ldots n-5}$ as we do so. We then use $A_{n-8\ldots n-5}$ (which we just inferred) and G to infer S_0 using point 2 of Section 5.4 above, which gives us our last four inputs $A_{n-4\ldots n-1}$.

5.6 Avalanche

This F function does not provide full avalanche between input and output. I see no secure way to build a balanced full-avalanche F function for large blocks that isn't so much slower than Mercy's F function that the time cost would not be better spent on more rounds with the weaker F function.

Instead of providing full avalanche, the F function makes two weaker guarantees that together are almost as effective:

- A change to any bit of the input will on average change half the bits of the last 128 bits of the output
- A change to any bit of the last 128 bits of the input will on average change half the bits of the output

A full avalanche F function achieves avalanche across the whole block after three rounds. This construction does so after four rounds. In addition, in encryption and decryption every keytable lookup depends on the result from the previous lookup.

The partial collision attack from Section 2 will demonstrate that after six rounds Mercy's avalanche is imperfect, since only 384 bits of state are carried between some parts of the block, but such an attack would require that 2^{193} chosen plaintexts be encrypted, and is thus outside the security goals of the cipher. This suggests a distinction between perfect avalanche, and avalanche sufficient to defeat cryptanalysis; this distinction is useful since mechanisms for providing perfect avalanche, such as networks based on Fourier transforms (used in SAFER [12] and proposed by [16] for large block ciphers), can carry a heavy performance penalty on large blocks. This distinction is not useful on small blocks: if this attack is possible against a cipher with a 64-bit block, it will not require more than 2^{33} chosen plaintexts.

5.7 Whitening

Mercy only whitens one half of the input and output, since the cost both in time and storage of whitening both halves would be significant, and since the primary function of whitening is to hide the input of the F function from attackers. On large block ciphers, whitening also serves to frustrate attacks based on creating inputs with special structures, such as attempts to induce a repeating state in the state machine of F.

5.8 Linear and differential cryptanalysis

We do not prove Mercy resistant to either linear or differential cryptanalysis; a large block cipher meeting the performance goals that could be proven resistant would be a worthy goal. However, four features of the cipher lend resistance to these classes of attack.

First, the key dependent subsitutions are optimised against linear and differential cryptanalysis as described in Section 5.2.

Second, the LFSR-based construction of the Q state machine forces any input to the F function with active substitutions (in the terminology of [7]) to make at least three substitutions active. In practice, the intent of the F function design is that any difference in the input causing a difference in a T box substitution will cause all subsequent T box subsitution to be uncorrelated; avoiding this effect will be very hard for attackers. Most F functions cannot afford the luxury of 68 chained non-linear substitutions.

Third, the initial and final whitening should force attackers to look for difference propagations or linear correlations over at least four rounds.

Fourth, the ways in which key material is introduced in the F function should mean that inferring a suggested key from a given plaintext-ciphertext pair should be extremely difficult.

6 Conclusions

Large blocks are useful in any application where random access to data is desirable, of which sector encryption is a prime example. Mercy is intended to demonstrate the possibility of building an efficient block cipher for large blocks. Mercy's design was inspired by two beliefs:

- Large block sizes can lend useful advantages in security and speed
- Avalanche across large blocks need not be perfect before a cryptanalyst with limited computing resources cannot distinguish it from perfect, as explained in Section 5.6.

However, the primary motivation for the design of Mercy is not that the cipher be directly used for this application; it is to inspire further work where it is badly needed. Disk encryption suffers none of the barriers to adoption from interoperability suffered by (for example) email encryption. But it is very rarely used, and the main barrier to adoption among security conscious users is the high performance penalties it currently exacts. I hope that Mercy demonstrates that fast, secure large block ciphers are possible, and inspires better cryptographers to propose better designs.

Acknowledgements

Thanks are due to the anonymous reviewers both on this submission and on submission to Crypto '99, who provided valuable references and suggestions, to Stefan Lucks for his sound advice, and to David Wagner, David A Molnar, Conrad Hughes, and Keith Lockstone for their comments on an earlier design and for encouragement. Special thanks to Scott Fluhrer who has found a very effective differential attack on the cipher presented here, details of which are on the Mercy home page.

References

1. Ross Anderson and Eli Biham. Two practical and provably secure block ciphers: BEAR and LION. In Gollman [8], pages 113–120.
2. Mihir Bellare and Phillip Rogaway. On the construction of variable-input-length ciphers. In Lars R. Knudsen, editor, *Fast Software Encryption: 6th International Workshop*, volume 1636 of *Lecture Notes in Computer Science*, pages 231–244, Rome, Italy, March 1999. Springer-Verlag.
3. Eli Biham, editor. *Fast Software Encryption: 4th International Workshop*, volume 1267 of *Lecture Notes in Computer Science*, Haifa, Israel, 20–22 January 1997. Springer-Verlag.
4. Eli Biham and Alex Biryukov. How to strengthen DES using existing hardware. In Josef Pieprzyk and Reihanah Safavi-Naini, editors, *Advances in Cryptology—ASIACRYPT '94*, volume 917 of *Lecture Notes in Computer Science*, pages 398–412, Wollongong, Australia, 28 November–1 December 1994. Springer-Verlag.
5. Craig S. K. Clapp. Optimizing a fast stream cipher for VLIW, SIMD, and superscalar processors. In Biham [3], pages 273–287.
6. Joan Daemen and Craig S. K. Clapp. Fast hashing and stream encryption with Panama. In Serge Vaudenay, editor, *Fast Software Encryption: 5th International Workshop*, volume 1372 of *Lecture Notes in Computer Science*, pages 60–74, Paris, France, 23–25 March 1998. Springer-Verlag.
7. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. NIST AES Proposal, 1998.
8. Dieter Gollman, editor. *Fast Software Encryption: Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, Cambridge, UK, 21–23 February 1996. Springer-Verlag.
9. Peter Gutmann. Secure filesystem. `http://www.cs.auckland.ac.nz/%7Epgut001/sfs/index.html`, 1996.
10. K. Kaukonen and R. Thayer. A stream cipher encryption algorithm "ARCFOUR". Internet-Draft draft-kaukonen-cipher-arcfour-03.txt, July 1999. The draft is a work in progress, but the algorithm (as RC4(tm)) is due to Ronald L. Rivest.
11. Stefan Lucks. BEAST: A fast block cipher for arbitrary blocksizes. In *FIP TC-6 and TC-11 Joint Working Conference on Communications and Multimedia Security*, September 1996.
12. James L. Massey. SAFER K-64: A byte-oriented block-ciphering algorithm. In Preneel [15]. Published 1995.
13. Mitsuru Matsui. New structure of block ciphers with provable security against differential and linear cryptanalysis. In Gollman [8], pages 205–218.
14. Chris J. Mitchell. Authenticating multicast Internet electronic mail messages using a bidirectional MAC is insecure. In *IEEE Transactions on Computers*, number 41, pages 505–507. 1992.
15. Bart Preneel, editor. *Fast Software Encryption: Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*, Leuven, Belgium, 14–16 December 1994. Springer-Verlag. Published 1995.

16. Terry Ritter. A mixing core for block cipher cryptography. `http://www.io.com/%7Eritter/MIXCORE.HTM`, 1998.
17. Phillip Rogaway and Don Coppersmith. A software-optimized encryption algorithm. In Ross Anderson, editor, *Fast Software Encryption*, pages 56–63. Springer-Verlag, 1994.
18. Bruce Schneier and Doug Whiting. Fast software encryption: Designing encryption algorithms for optimal software speed on the Intel Pentium processor. In Biham [3], pages 242–259.
19. Rich Schroeppel. Hasty Pudding Cipher specification. NIST AES Proposal, June 1998.
20. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
21. David Wheeler. A bulk data encryption algorithm. In Preneel [15]. Published 1995.