

Turing: a fast stream cipher

Greg Rose, Philip Hawkes

QUALCOMM Australia, Level 3, 230 Victoria Road, Gladesville NSW 2111, Australia
{ggr, phawkes}@qualcomm.com

Abstract. This paper proposes the Turing stream cipher. Turing offers up to 256-bit key strength, and is designed for extremely efficient software implementation. It combines an LFSR generator based on that of SOBER[27] with a keyed mixing function reminiscent of a block cipher round. Aspects of the block mixer round have been derived from Rijndael[20], Twofish[21], tc24[23] and SAFER[22].

1. Introduction

Turing (named after Alan Turing) is a stream cipher designed to simultaneously be:

- Extremely fast in software on commodity PCs
- Usable in very little RAM on embedded processors
- Exploit parallelism to enable fast hardware implementation.

The Turing stream cipher has a major component, the Linear Feedback Shift Register, which originated with the design of SOBER [17-19,25-27]. Analyses of the SOBER family are found in [2,3,4,5,9,10]. The S-box used in Turing is partially derived from that used in SOBER-t32 and is described in [7]. The efficient LFSR updating method is modeled after that of SNOW 2.0 [24].

Turing combines the LFSR generator with a keyed mixing function reminiscent of a block cipher round. Aspects of the block mixing function have been derived from Rijndael[20], Twofish[21], tc24[23] and SAFER[22].

Turing is designed to meet the needs of embedded applications such as voice encryption in wireless telephones that place severe constraints on the amount of processing power, program space and memory available for software encryption algorithms. Since most of the mobile telephones in use incorporate a microprocessor and memory, a software stream cipher that is fast and uses little memory would be ideal for this application. Turing overcomes the inefficiency of binary LFSRs in a manner similar to SNOW 2.0[24] by utilizing an LFSR defined over $GF((2^8)^4)$ (a different isomorphic representation of $GF(2^{32})$) and a number of techniques to greatly increase the generation speed of the pseudo-random stream in software on a general processor. Turing allows an implementation tradeoff between small memory use, or very high speed using pre-computed tables. Reference source code showing small memory, key agile, and speed-optimized implementations is available at [29].

Turing has four components: key loading, Initialisation vector (IV) loading, an LFSR, and a keyed non-linear filter (NLF). A block diagram of the latter two is shown in Figure 1.

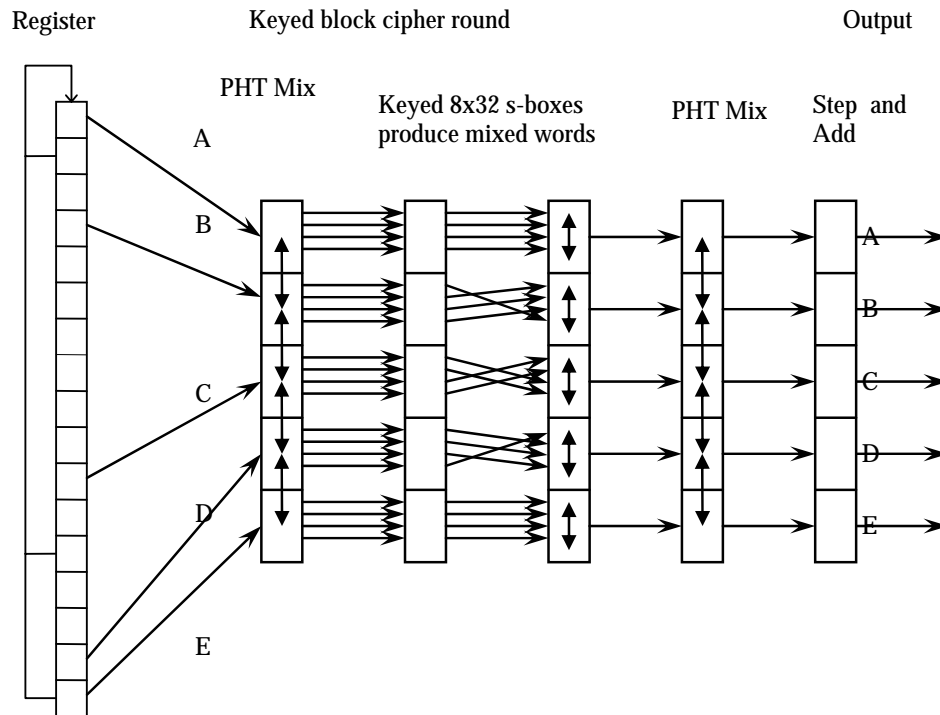


Figure 1. The components of the Turing stream cipher.

Five words selected from the LFSR are first mixed, then passed through a highly nonlinear S-box transformation, mixed again, and combined with four new words directly from the LFSR, to create 160 bits of keystream.

Byte Ordering Considerations

Turing utilizes native processor operations on word-sized data items, but is expected to accept keys that are simply strings of bytes, and to produce a stream of bytes as output for encryption purposes. This means that a translation between native byte ordering and external byte ordering is necessary to ensure compatibility between implementations running on different processors. Since all Internet standards are defined using “big-endian” byte ordering, in which the most significant byte of a multi-byte quantity appears first in memory, this is what is chosen for Turing. On “little-endian” machines, the bytes of the key and IV must be assembled into words,

and the words of the output stream must be byte reversed before being XORed into the buffer.

Note that it is simple to define a cipher that is exactly equivalent to Turing except that it is “little-endian”. This cipher would share all the security aspects of the original, but might execute a bit more efficiently on such CPUs. (In practice, compilers often recognize the idiom to do the byte-swapping and generate extremely efficient code; we could not measure any difference in execution time between raw and byte-swapped versions on either Sun Ultra-SPARC or mobile Pentium III processors.)

The paper is set out as follows. First, the LFSR is defined in Section 2. Section 3 describes the NLF and explains how the overall structure of Turing operates. The key and initialization vector loading is described in Section 4. Section 5 discusses performance, and Section 6 analyses security and possible attacks.

2. LFSR of Turing

2.1 Linear Feedback Shift Register Generalities

Binary Linear Feedback shift registers can be extremely inefficient in software on general-purpose microprocessors. LFSRs can operate over any finite field, and can be made more efficient in software by utilizing a finite field more suited to the processor. Particularly good choices for such a field are the Galois Field with 2^w elements ($GF(2^w)$), where w is related to the size of items in the underlying processor, usually bytes or 32-bit words. The elements of this field and the coefficients of the recurrence relation occupy exactly one unit of storage and can be efficiently manipulated in software. In the meantime, the order k of the recurrence relation that encodes the same amount of state is reduced by a factor of w .

The field $GF(2^w)$ can be represented (the *standard representation*) as the modulo 2 coefficients of all polynomials with degree less than w . That is, an element a of the field is represented by a w -bit word with bits $(a_{w-1}, a_{w-2}, \dots, a_1, a_0)$, which represents the polynomial

$$a_{w-1}x^{w-1} + a_{w-2}x^{w-2} + \dots + a_1x + a_0.$$

Turing takes this a step further, using 8-bit bytes to represent elements of $GF(2^8)$, and 32-bit words to represent degree-3 polynomials of bytes. The LFSR consists of 17 words of state information. Thus $w = 32$.

The LFSR is mathematically equivalent to w parallel bit-wide shift registers over $GF(2)$ each of length equivalent to the total state $17w$, each with the same recurrence relation but different initial state [10]. Let the polynomial $p_1(x)$ represent the LFSR

over $GF(2)$ (See Appendix D for $p_1(x)$). The configuration is chosen to make updating the LFSR as efficient as possible, subject to the following constraints:

- **The LFSR has maximum length period.** The period has a maximum length of $2^{17w}-1$ when $p(x)$ is a *primitive* polynomial of degree 17.
- **Approximately half of the coefficients of $p_1(x)$ are 1.** This condition is ideal for maximum diffusion and strength against cryptanalysis.

2.2 Specifics of the LFSR

B , the *bytes*, represents the Galois finite field $GF(2^8)$ represented modulo the irreducible polynomial $z^8 + \gamma_7z^7 + \gamma_6z^6 + \gamma_5z^5 + \gamma_4z^4 + \gamma_3z^3 + \gamma_2z^2 + \gamma_1z + \gamma_0$, where the γ_i are bits, specifically $z^8 + z^6 + z^3 + z^2 + 1$. The constant $\beta_0 = 0x67$ below represents the polynomial $z^8 + z^7 + z^2 + z + 1$ for example.

W , the *words*, represents the Galois finite field $GF(B^4)$ represented modulo the irreducible polynomial $y^4 + \beta_3y^3 + \beta_2y^2 + \beta_1y + \beta_0$, where the $\beta_i \in B$ are *bytes*, specifically $y^4 + 0xD0.y^3 + 0x2B.y^2 + 0x43.y + 0x67$. The element α used below is the polynomial y .

The Turing LFSR, then, consists of 17 32-bit *words*, with characteristic polynomial $x^{17} + x^{15} + x^4 + \alpha$, where $\alpha \in W$ is the polynomial y .

The equivalent binary polynomial is shown in Appendix D.

Having defined all of that, implementation of the LFSR can be done very efficiently, because the constant α is so simple. Multiplication by α consists of shifting the word left by 8 bits, and adding (XOR) a precomputed constant from a table indexed by the most significant 8 bits. In C, calculating the new word to be inserted in the LFSR is:

```
new = R[15] ^ R[4] ^ (R[0] << 8) ^ Multab[R[0] >> 24];
```

with the precomputed table shown in Appendix A.

The LFSR can then be updated by:

```
R[0] = R[1]; R[1] = R[2]; ... ; R[16] = R[17]; R[17] = new;
```

After updating the LFSR, 5 words from its state are selected for input to the nonlinear filter. These are r_{16} , r_{13} , r_6 , r_1 , r_0 , referred to as A , B , C , D , E (respectively) below. These tap positions form a “*full positive difference set*”, so that as words move through the register and are selected as input to the nonlinear filter function, no pair of words is used more than once [8]. For each block of output produced, the shift register is stepped twice, with elements drawn from the same positions, so 9 of the 17 elements are used (element 0 after the first step becomes element 1 after the second step).

3. The Nonlinear Filter

The only component of Turing that is explicitly nonlinear is its S-boxes. Additional nonlinearity also comes from the combination of the operations of addition modulo 2^{32} and XOR; while each of these operations is linear in its respective mathematical

group, each is slightly nonlinear in the other's group. The nonlinear filter in Turing consists of:

- Stepping the LFSR and selecting the 5 input words
- Mixing the words using the Pseudo-Hadamard Transform¹
- Transforming the bytes with keyed transformations, and mixing the words using four 8→32 bit nonlinear S-boxes
- Again mixing the words using the Pseudo-Hadamard Transform
- Stepping the LFSR again and adding (mod 2^{32}) more input words

3.1 The Key-Dependent S-box Transformation

Turing transforms each word using four logically independent 8→32 S-boxes applied to each byte of the input word and XORed, in a manner similar to that used in Rijndael[20]. However, unlike Rijndael, this transformation is in general not invertible, as the expansion to 32 bits is nonlinear.

These S-boxes are based in turn on a fixed 8→8 bit permutation S-box and a fixed nonlinear 8→32 bit *Qbox*, iterated with the data modified by variables derived during key setup.

The words *B*, *C* and *D* are rotated left by 8, 16 and 24 bits respectively, before the S-box transformation, to address a potential attack described below.

3.1.1 Derivation of the Sbox

The fixed S-box is referred to in the rest of this document as *Sbox*[.]. It is a permutation of the input byte, and has a minimum nonlinearity of 104, and is shown in Appendix B. The S-box shown was derived by the following procedure, based on the well-known stream cipher RC4™. RC4 was keyed with the 11-character ASCII string "Alan Turing", and then 256 generated bytes were discarded. Then the current permutation used in RC4 was tested for nonlinearity, another byte generated, etc., until a total of 10000 bytes had been generated. The best observed minimum nonlinearity was 104, which first occurred after 736 bytes had been generated. The corresponding state table, that is its internal permutation after keying and generating 736 bytes, forms *Sbox*. By happy coincidence it also has no fixed points (ie. $\forall x, Sbox[x] \neq x$).

3.1.2 Derivation of the Qbox

The *Qbox* is a fixed nonlinear 8→32-bit table. It was developed by the Queensland University of Technology at our request[7]. It is best viewed as 32 independent Boolean functions of the 8 input bits. The criteria for its development were:

- the functions should be highly nonlinear (each has nonlinearity of 114)
- the functions should be balanced
- the functions should be pairwise uncorrelated

¹ Calling this a "Pseudo-Hadamard Transform" is a bit of a stretch, but we couldn't think of a better term for it. Some documents call it *n-PHT*.

3.1.3 The Key Dependent Sboxes

Turing uses four keyed 8→32 bit S-boxes S_i which can be calculated when required for small memory or key-agile implementation, or pre-calculated at key setup time for high throughput implementation. The fixed Sbox and Qbox together require 1280 bytes of read-only memory. If the keyed S-boxes are pre-calculated they require 4k of memory. Each data word is broken into bytes, and the outputs of the four S-boxes applied to the bytes are XORed.

The key material is accessed as words k_j , $0 \leq j < N$, where N is the number of words in the key. Section 4.1 below describes the key scheduling process.

S_i ($0 \leq i \leq 3$) is derived from the corresponding byte positions of the scheduled key. The input byte is combined with a key byte and passed through the *Sbox*, the result is combined with another key byte, and so on, to form a temporary result.

$$t_i(x) = \text{Sbox}[K_{i,N-1} \oplus \text{Sbox}[K_{i,N-2} \oplus \dots \text{Sbox}[K_{i,0} \oplus x] \dots]]$$

where \oplus is the XOR operator, and N is the number of words in the key. Note that $t_i(.)$ forms a permutation. This process can be visualised as the input byte “bouncing around” under the control of the key. At each bounce, a rotated word from the Qbox is accumulated into another temporary word w ; the rotation depends on the byte position in question and the stage of progress, ensuring that no entries of the Qbox can cancel each other out.

The accumulated word w is highly nonlinear with respect to the input, and highly dependent on the key material, however the bit positions in it are not likely to be balanced. The byte t , being a permutation, is by definition balanced. So replacing the corresponding byte of w with t forms the final word for this input byte. The following C code illustrates this process for a single byte position (position 0, most significant byte):

```
WORD S0(BYTE b)
{
    int          i;
    WORD         ws;

    ws = 0;
    for (i = 0; i < keylen; ++i) {
        b = Sbox[B(K[i], 0) ^ b];
        ws ^= ROTL(Qbox[b], i + 0); /* "+0" for MSB */
    }
    ws = (ws & 0x0FFFFFFUL) | (b << 24);
    return ws;
}
```

3.4 The “Pseudo-Hadamard Transform” (PHT)

In the cipher family of SAFER, Massey uses this very simple construct to mix the values of two bytes: $(a, b) = (2a+b, a+b)$, where the addition operation is addition modulo 2^8 , the size of the bytes. In Turing, the operation is extended to mix words, using addition modulo 2^{32} , and is further extended to mix an arbitrary number of words. As an example, 5 words *A..E* are mixed by the matrix multiplication:

$$\begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix}$$

Note that all diagonal entries are 2 except the last diagonal entry is 1, not 2. In C, this is easily implemented:

```
E = A + B + C + D + E;
A += E; B += E; C += E; D += E;
```

This extended PHT construction is due to Tom St Denis, used in his *tc24* block cipher [23], although others have apparently done the same thing.

3.5 The Final Addition Stage

The first four operations of the nonlinear filter are not invertible. In particular, not all 160-bit results are possible, and the results produced are not equally likely. Adding more input words to the output has three effects: first, it makes all output values possible and equally likely²; second, these operations “lock in” the mixing of the last PHT stage, since an attacker needs to remove the effects of these words before being able to reverse these mixing rounds; finally, by adding four new words, approximately half of the register state is involved in the filter function.

3.5 Output

The five words produced by this processing are used as the output keystream, in the order *A..E*, most significant byte of each word first. Issues of buffering bytes to encrypt data that is not aligned as multiples of 20 bytes are considered outside the scope of this document.

4 Keying the Stream Cipher

For Turing, the key and Initialization Vector (IV) are presented as a byte stream, and converted to 32-bit words in the most significant byte first (big-endian) representation. The key and IV must therefore be multiples of 4 bytes each.

The minimum size of the key is 32 bits; although clearly this is useless cryptographically, Turing makes a reasonably good pseudorandom number generator

² This would be true if the new words were completely independent of the original input words, however one word is reused. It's very close to true, though.

for statistical and simulation purposes. We hope for security equal to key enumeration for keys up to 256 bits. The largest key size supported is 256 bits. There is usually a key-loading stage, but for embedded applications with a key fixed in read-only memory, it is permissible for a high quality cryptographic key to be used directly as if it was the output of the key loading stage.

The minimum size of the IV is zero, however an IV loading stage is mandatory even in this case, because the LFSR is initialized when the IV is loaded. The maximum size of the IV is determined by the key length, such that the sum of the key length and IV length is no more than 12 words (384 bits). There is no requirement that the size of the IV be constant.

The structure of Turing guarantees that different key/IV length combinations will generate distinct output streams. No more than 2^{160} (160-bit) blocks of output should be generated using any one key/IV combination. (We don't consider this to be a meaningful limitation, nor will we consider distinguishing attacks that require more than this amount of known plaintext to be a weakness in Turing.)

4.1 Key Loading

Turing's key loading process mixes the key bytes through the fixed S-box and the Q-box, to ensure that all bytes of the key affect all four of the keyed S-boxes. For each word of the key, the bytes are transformed serially through the S-box, using the Q-box in an unbalanced Feistel structure for each byte to alter the other three bytes of the word. After this, the words are mixed using the extended PHT transform. Thus the transformation is reversible, ensuring that no keys are equivalent. The resulting words are stored for subsequent use; they occupy the same amount of space as the original key, which is no longer needed. These words will be used in the key-dependent S-boxes, and also during the IV loading process to initialize the LFSR.

Note that the only reason for the key transformation is to thwart the somewhat unlikely related key attack; without some sort of mixing, similar keys would produce similar key-dependent S-boxes. With this mixing, an attacker would need to know the key to be able to predict or minimize the effect of a related key. This is why we allow the possibility that the processed key can be provisioned directly into hardware.

The bytes K_{ij} mentioned above are the bytes of these stored words; the j index ($0 \leq j < N$, where N is the number of words of the key) locates the word of the stored mixed key, while the i index ($0 \leq i \leq 3$) is the byte of the word, with the byte numbered 0 being the most significant byte.

If the fastest implementation of Turing is desired, at this point the effect of the keyed S-boxes can be computed into four tables, each with 256 32-bit entries. The combined operations then consist of four byte-index table lookups and four word XOR operations for each input word. A similar optimization is used in fast implementations of Rijndael.

4.2 IV loading

The Initialization Vector loading process initializes the LFSR with values derived from a non-linear mixing of the key and the IV. The LFSR is initialized with words in the following manner:

- The IV words are copied into place and processed using the same invertible S-box transformation mentioned in the preceding section
- The key words are appended, without further processing
- Let L be the length (in words) of the key, and I be the length in words of the IV. A single word, $0x010203LI$ is appended. This ensures that different length keys and IVs cannot create the same initial LFSR state.
- The remaining words of the register are filled by adding the immediately previous word to the word $(L+I)$ before that, then processing the resulting word with the keyed S-box. That is, the k th word K_k ($L+I+1 \leq k < 17$) is set to $S[K_{k-1}+K_{k-L-I-1}]$.
- Finally, once the LFSR has been filled with data, the contents are mixed with a 17-word-wide PHT. Keystream generation can now begin.

5. Performance

If sufficient random-access memory is available, the operations of the four keyed S-boxes can be precalculated at the time of key setup, resulting in four tables, one for each byte of the input word.

Many current high-end microprocessor CPUs allow multiple instructions to execute at once, if the instructions are sufficiently independent. Note that the operations mentioned above are all highly parallel, allowing very good performance on such processors. Similarly hardware or FPGA implementations can achieve high throughput using parallel paths.

In the cases where the key is provisioned into hardware, it is possible for the entire key scheduling process, including the calculation of these tables, to be done at the time of provisioning; thus, instead of 4K bytes of RAM and 1280 bytes of ROM, 4K bytes of ROM is sufficient and yields a very fast implementation. (A further 1024 bytes of ROM is still required for the multiplication table.)

Lastly, note that there is no accumulation of nonlinear data, nor is the clocking irregular. Therefore, if it is desirable to generate a small amount of keystream offset in a much larger block, this can be done by “fast forwarding” the LFSR using polynomial or matrix exponentiation in logarithmic time, rather than the linear time that would be required to generate and discard the intermediate output.

Turing provides flexibility of efficient implementation. The source code archive [29] includes 4 separate implementations. These are:

- TuringRef.c, an unoptimized reference implementation, which also uses little Random Access Memory. It does not precompute any tables.
- TuringTab.c precomputes the tables required by the keyed S-boxes when the key is set. It uses 4K bytes of RAM in addition to the LFSR.

- TuringLazy.c is a key-agile implementation, which fills in entries of the S-box tables only as they are required (lazy evaluation). Thus key and IV setup are relatively fast, and encryption speed is adequate.
- TuringFast.c uses S-box tables computed at key setup time, and performs as much computation inline as possible.

The following table shows various performance figures. These are measured times on an IBM laptop with 900MHz Pentium III processor, using Microsoft Visual C++ V6.0, with the optimization options for "Release" build. Comparison times for Brian Gladman's (highly optimized) implementation of AES and our implementation of an RC4 compatible cipher with a bulk encryption interface are also shown.

Cipher	cycles/B	Key	IV setup	tables	RAM	MByte/s
TuringRef	134.92	477.00	4272.31	2304	68	6.67
TuringLazy	20.73	1802.70	991.80	2304	4164	43.42
TuringTab	17.26	72457.93	900.90	2304	4164	52.15
TuringFast	5.45	72417.12	882.90	2304	4164	165.15
arsyfor	37.49	0.00	10347.42	0	258	24.00
AES enc.	26.85	239.00	0.00	20480	176	33.53
MHz	900.00					

Notes: All figures are for 128-bit keys. We consider RC4's keying operation to actually be "IV setup", and this does not include time to either discard generated bytes or to hash the key and IV, which would be necessary for security.

6. Security

In this section, we attempt to justify Turing's security by reference to the mechanisms by which it defeats a variety of known attacks. The underlying philosophy of Turing is to combine two independently strong mechanisms in a manner that allows each one to protect the other against standard attacks. The two mechanisms used are the nonlinear filter generator structure, in combination with the highly nonlinear, noninvertible key dependent S-box transformation. The security of Turing relies on both of these components.

6.1 Known Plaintext

Turing is a synchronous stream cipher, so the keystream generated is independent of the plaintext. Misuse of any stream cipher, such as reusing keystream, can result in compromise of the plaintext without actually revealing any information about the cipher generator itself. Turing increases safety by having an integrated mechanism to support Initialization Vectors making it easier to use correctly.

To attack the stream generator itself requires significant knowledge about the plaintext, either completely known plaintext or at least significant redundancy in the

input language. In the discussion below, we will assume known or chosen plaintext, or equivalently that the attacker has direct access to the stream generator output.

6.2 Statistical Attacks

A keystream generator that exhibits basic statistical biases or detectable characteristics is weak. The LFSR used has well studied statistical properties that translate directly to the output. Additionally the highly nonlinear transformation in the core of Turing serves to disguise the inherent linearity of the LFSR output.

We have extensively tested output from Turing using the Crypt-X package[6] and have detected no statistical weaknesses.

6.3 Related Key and Related / Chosen Initialization Vector Attacks

Related key attacks assume that the attacker can somehow obtain output from a black box whose key is closely related to that of the cipher instance being attacked. Turing's key loading mechanism exists solely to address this attack, by ensuring that a change to any single byte of the key will significantly (and nonlinearly) alter the behavior of all the S-boxes and also the initial loading of the LFSR. The transformation performed is bijective and publicly known, so it is easy to create pairs of input keys which are very similar after transformation, however finding a key which is similar to an unknown key appears difficult.

Initialization Vectors are often related (e.g. counters are often used) and might even be chosen by the attacker. The IV is used to initialize the LFSR, so we have been careful to fill the LFSR in a highly key-dependent and nonlinear manner. Any change in the IV first makes a large change in the corresponding word loaded. That word will cause an unpredictable change in at least one of the fill words, then those changes will be propagated through the LFSR with the PHT transform. LFSR states derived from different IVs are less obviously related than states drawn from different segments of the same output stream.

6.4 Correlation and Distinguishing Attacks

Coppersmith et. al. have defined a fairly general model [28] for Distinguishing Attacks against nonlinear filter generators. The model assumes that some significant correlation can be identified in the filter function, and that this correlation will remain usable after outputs have been combined in such a way as to eliminate the linear part from consideration.

Turing's nonlinear filter has been defined to use a significant amount of input state, and to perform a strong transformation of it. While some correlations must by definition exist, and it is our hope that they might be sufficiently small to preclude this kind of attack, we do not rely on this hope for security. Instead, the fact that the S-boxes are dependent on the secret key makes this kind of attack inapplicable.

In turn, the unknown values being generated in the LFSR form a kind of "whitening" to protect the nonlinear filter function from analysis.

6.5 Nonlinear Algebraic Attacks

In [30] Courtois gives guidelines for nonlinear filter generators to be safe against higher order algebraic attacks. We are still performing detailed analysis of typical S-boxes used by Turing, but generally speaking the nonlinear functions are complex and of high degree and each involve at least 8 intermediate binary variables. Additionally, the fact that the functions are key-dependent would appear to make this avenue of attack inapplicable.

6.6 Guess and Determine Attacks

Guess and Determine attacks proceed by using linear and nonlinear relations to allow some state words to be guessed and others determined from them. The choice of feedback and output positions in Turing is copied from the SOBER t-class ciphers[20]. This structure was mechanically optimized against these kinds of attacks, and has been extensively analyzed for the NESSIE project, and should provide a minimum complexity exceeding the enumeration of 256-bit keys. In addition, the attacks rely upon the fact that the nonlinear function can be rewritten so that given its output, and $n-1$ of its n inputs, the remaining input can be determined. Turing's nonlinear filter function design frustrates this, by (a) being key-dependent, (b) being noninvertible, and (c) requiring a large amount of output to build a large inversion table.

6.7 Differential Trail

The main advantage of the PHT construct is its speed and parallelism, however it is not as good a mixing function as could be desired. One undesirable characteristic is that if two of the input words from A , B , C , or D are equal, they remain equal after the transformation. The S-boxes operate only on individual words, and so also preserve this equality. This allowed an attack where a 64-bit assumption yielded 96 linear equations relating to the state of the register. Even though such this attack appeared to have complexity greater than 256-bit exhaustive search, this undesirable differential characteristic is addressed by rotating the input to the S-boxes corresponding to B , C , and D , making this advantage negligible.

7. Acknowledgements

The authors would like to thank Thomas St. Denis, Scott Fluhrer, David McGrew and David Wagner for useful insights and feedback into the design of Turing.

References

1. S. Blackburn, S. Murphy, F. Piper and P. Wild, "A SOBERing Remark". Unpublished report. Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U. K., 1998.

2. S. Blackburn, S. Murphy, F. Piper and P. Wild, "Brief Comments on SOBER-II". Unpublished report. Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U. K., 1998.
3. D. Bleichenbacher and S. Patel, "SOBER cryptanalysis", Pre-proceedings of *Fast Software Encryption '99*, 1999, pp. 303-314.
4. D. Bleichenbacher, S. Patel and W. Meier, "Analysis of the SOBER stream cipher". TIA contribution TR45.AHAG/99.08.30.12.
5. E. Dawson, B. Millan and L. Simpson, "Security Analysis of SOBER". Unpublished report by the Information Systems Research Centre, Queensland University of Technology, 1998.
6. E. Dawson, A. Clark, H. Gustafson and L. May, "CRYPT-X'98, (Java Version) User Manual", *Queensland University of Technology*, 1999.
7. E. Dawson, W. Millan, L. Burnett, G. Carter, "On the Design of 8×32 S-boxes". Unpublished report, by the Information Systems Research Centre, Queensland University of Technology, 1999.
8. J. Dj. Golić, "On Security of Nonlinear Filter Generators", *Proc. Fast Software Encryption 1996 Cambridge Workshop*, Springer-Verlag 1996.
9. C. Hall and B. Schneier, "An Analysis of SOBER", 1999.
10. P. Hawkes, "An Attack on SOBER-II". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1999. See <http://www.home.aone.net.au/qualcomm>.
11. See T. Herlestam, "On functions of Linear Shift Register Sequences", in Franz Pichler, editor, *Proc. EUROCRYPT 85*, LNCS 219, Springer-Verlag 1986.
12. G. Marsaglia, "DIEHARD", <http://stat.fsu.edu/~geo/diehard.html>
13. A. Menezes, P. Van Oorschot, S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1997, Ch 6.
14. C. Paar, Ph.D. Thesis, "Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields", *Institute for Experimental Mathematics, University of Essen*, 1994, ISBN 3-18-332810-0.
15. B. Schneier, "Applied Cryptography Second Edition", *Wiley 1996*, pp. 369-413.
16. TIA/EIA Standard IS-54B, Telecommunications Industry Association, Vienna VA., USA.
17. G. Rose, "A Stream Cipher based on Linear Feedback over $GF(2^8)$ ", in C. Boyd, Editor, *Proc. Australian Conference on Information Security and Privacy*, Springer-Verlag 1998.
18. G. Rose, "SOBER: A Stream Cipher based on Linear Feedback over $GF(2^8)$ ". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1998. See <http://people.qualcomm.com/ggr/QC>. (expanded version of [17]).
19. G. Rose, "S32: A Fast Stream Cipher based on Linear Feedback over $GF(2^{32})$ ". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1998. See <http://people.qualcomm.com/ggr/QC>.

20. J. Daemen and V. Rijmen. "AES Proposal: Rijndael".
www.esat.kuleuven.ac.be/~rijmen/rijndael/
21. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-Bit Block Cipher. In Selected Areas in Cryptography '98, June 1998. Lecture Notes in Computer Science.
22. James L. Massey, "{SAFER} K-64: A Byte-Oriented Block-Ciphering Algorithm", in Proc. Fast Software Encryption, Lecture Notes in Computer Science, 1993.
23. Thomas St Denis, "weekend cipher", sci.crypt news article
3d4d614d_17@news.teranews.com
24. Patrick Ekdahl, Thomas Johansson, "A new version of the stream cipher SNOW", Proc. Selected Areas in Cryptography, LNCS 2002.
25. Philip Hawkes, Greg Rose, "The t-class of SOBER Stream Ciphers",
<http://people.qualcomm.com/ggr/QC/tclass.pdf>
26. Philip Hawkes, Greg Rose, "SOBER-t16", submission to NESSIE
27. Philip Hawkes, Greg Rose, "SOBER-t32", submission to NESSIE
28. Coppersmith et al, "Cryptanalysis of Stream Ciphers with Linear Masking", proc. Crypto 2002, LNCS 2442, Springer 2002.
29. Reference source code for Turing, <http://people.qualcomm.com/ggr/QC/Turing.tgz>
30. Nicolas T. Courtois, "Higher Order Correlation Attacks, XL algorithm and Cryptanalysis of Toyocrypt" not yet published.

Appendix A: Multiplication Table for Turing.

```

/* Multiplication table for Turing using 0x72C688E8 */
unsigned long Multab[256] = {
    0x00000000, 0x72C688E8, 0xE4C15D9D, 0x9607D575,
    0x85CFBA77, 0xF709329F, 0x610EE7EA, 0x13C86F02,
    0x47D339EE, 0x3515B106, 0xA3126473, 0xD1D4EC9B,
    0xC21C8399, 0xB0DA0B71, 0x26DDDE04, 0x541B56EC,
    0x8EEB7291, 0xFC2DFA79, 0x6A2A2F0C, 0x18ECA7E4,
    0x0B24C8E6, 0x79E2400E, 0xEFE5957B, 0x9D231D93,
    0xC9384B7F, 0xBBFEC397, 0x2DF916E2, 0x5F3F9E0A,
    0x4CF7F108, 0x3E3179E0, 0xA836AC95, 0xDAF0247D,
    0x519BE46F, 0x235D6C87, 0xB55AB9F2, 0xC79C311A,
    0xD4545E18, 0xA692D6F0, 0x30950385, 0x42538B6D,
    0x1648DD81, 0x648E5569, 0xF289801C, 0x804F08F4,
    0x938767F6, 0xE141EF1E, 0x77463A6B, 0x0580B283,
    0xDF7096FE, 0xADB61E16, 0x3BB1CB63, 0x4977438B,
    0x5ABF2C89, 0x2879A461, 0xBE7E7114, 0xCCB8F9FC,
    0x98A3AF10, 0xEA6527F8, 0x7C62F28D, 0x0EA47A65,

```

```

0x1D6C1567, 0x6FAA9D8F, 0xF9AD48FA, 0x8B6BC012,
0xA27B85DE, 0xD0BD0D36, 0x46BAD843, 0x347C50AB,
0x27B43FA9, 0x5572B741, 0xC3756234, 0xB1B3EADC,
0xE5A8BC30, 0x976E34D8, 0x0169E1AD, 0x73AF6945,
0x60670647, 0x12A18EAF, 0x84A65BDA, 0xF660D332,
0x2C90F74F, 0x5E567FA7, 0xC851AAD2, 0xBA97223A,
0xA95F4D38, 0xDB99C5D0, 0x4D9E10A5, 0x3F58984D,
0x6B43CEA1, 0x19854649, 0x8F82933C, 0xFD441BD4,
0xEE8C74D6, 0x9C4AFC3E, 0x0A4D294B, 0x788BA1A3,
0xF3E061B1, 0x8126E959, 0x17213C2C, 0x65E7B4C4,
0x762FDBC6, 0x04E9532E, 0x92EE865B, 0xE0280EB3,
0xB433585F, 0xC6F5D0B7, 0x50F205C2, 0x22348D2A,
0x31FCE228, 0x433A6AC0, 0xD53DBFB5, 0xA7FB375D,
0x7D0B1320, 0x0FCD9BC8, 0x99CA4EBD, 0xEB0CC655,
0xF8C4A957, 0x8A0221BF, 0x1C05F4CA, 0x6EC37C22,
0x3AD82ACE, 0x481EA226, 0xDE197753, 0xACDFFFBB,
0xBF1790B9, 0xCDD11851, 0x5BD6CD24, 0x291045CC,
0x09F647F1, 0x7B30CF19, 0xED371A6C, 0x9FF19284,
0x8C39FD86, 0xFEFF756E, 0x68F8A01B, 0x1A3E28F3,
0x4E257E1F, 0x3CE3F6F7, 0xAAE42382, 0xD822AB6A,
0xCBEAC468, 0xB92C4C80, 0x2F2B99F5, 0x5DED111D,
0x871D3560, 0xF5DBBD88, 0x63DC68FD, 0x111AE015,
0x02D28F17, 0x701407FF, 0xE613D28A, 0x94D55A62,
0xC0CE0C8E, 0xB2088466, 0x240F5113, 0x56C9D9FB,
0x4501B6F9, 0x37C73E11, 0xA1C0EB64, 0xD306638C,
0x586DA39E, 0x2AAB2B76, 0xBCACFE03, 0xCE6A76EB,
0xDDA219E9, 0xAF649101, 0x39634474, 0x4BA5CC9C,
0x1FBE9A70, 0x6D781298, 0xFB7FC7ED, 0x89B94F05,
0x9A712007, 0xE8B7A8EF, 0x7EB07D9A, 0x0C76F572,
0xD686D10F, 0xA44059E7, 0x32478C92, 0x4081047A,
0x53496B78, 0x218FE390, 0xB78836E5, 0xC54EBE0D,
0x9155E8E1, 0xE3936009, 0x7594B57C, 0x07523D94,
0x149A5296, 0x665CDA7E, 0xF05B0F0B, 0x829D87E3,
0xAB8DC22F, 0xD94B4AC7, 0x4F4C9FB2, 0x3D8A175A,
0x2E427858, 0x5C84F0B0, 0xCA8325C5, 0xB845AD2D,
0xEC5EFBC1, 0x9E987329, 0x089FA65C, 0x7A592EB4,
0x699141B6, 0x1B57C95E, 0x8D501C2B, 0xFF9694C3,
0x2566B0BE, 0x57A03856, 0xC1A7ED23, 0xB36165CB,
0xA0A90AC9, 0xD26F8221, 0x44685754, 0x36AEDFBC,
0x62B58950, 0x107301B8, 0x8674D4CD, 0xF4B25C25,
0xE77A3327, 0x95BCBDCF, 0x03BB6EBA, 0x717DE652,
0xFA162640, 0x88D0AEA8, 0x1ED77BDD, 0x6C11F335,
0x7FD99C37, 0x0D1F14DF, 0x9B18C1AA, 0xE9DE4942,
0xBDC51FAE, 0xCF039746, 0x59044233, 0x2BC2CADB,
0x380AA5D9, 0x4ACC2D31, 0xDCCBF844, 0xAE0D70AC,
0x74FD54D1, 0x063BDC39, 0x903C094C, 0xE2FA81A4,
0xF132EEA6, 0x83F4664E, 0x15F3B33B, 0x67353BD3,
0x332E6D3F, 0x41E8E5D7, 0xD7EF30A2, 0xA529B84A,

```

```

    0xB6E1D748, 0xC4275FA0, 0x52208AD5, 0x20E6023D,
};

```

Appendix B: the Sbox

```

unsigned char Sbox[256] = {
    0x61, 0x51, 0xeb, 0x19, 0xb9, 0x5d, 0x60, 0x38,
    0x7c, 0xb2, 0x06, 0x12, 0xc4, 0x5b, 0x16, 0x3b,
    0x2b, 0x18, 0x83, 0xb0, 0x7f, 0x75, 0xfa, 0xa0,
    0xe9, 0xdd, 0x6d, 0x7a, 0x6b, 0x68, 0x2d, 0x49,
    0xb5, 0x1c, 0x90, 0xf7, 0xed, 0x9f, 0xe8, 0xce,
    0xae, 0x77, 0xc2, 0x13, 0xfd, 0xcd, 0x3e, 0xcf,
    0x37, 0x6a, 0xd4, 0xdb, 0x8e, 0x65, 0x1f, 0x1a,
    0x87, 0xcb, 0x40, 0x15, 0x88, 0x0d, 0x35, 0xb3,
    0x11, 0x0f, 0xd0, 0x30, 0x48, 0xf9, 0xa8, 0xac,
    0x85, 0x27, 0x0e, 0x8a, 0xe0, 0x50, 0x64, 0xa7,
    0xcc, 0xe4, 0xf1, 0x98, 0xff, 0xa1, 0x04, 0xda,
    0xd5, 0xbc, 0x1b, 0xbb, 0xd1, 0xfe, 0x31, 0xca,
    0xba, 0xd9, 0x2e, 0xf3, 0x1d, 0x47, 0x4a, 0x3d,
    0x71, 0x4c, 0xab, 0x7d, 0x8d, 0xc7, 0x59, 0xb8,
    0xc1, 0x96, 0x1e, 0xfc, 0x44, 0xc8, 0x7b, 0xdc,
    0x5c, 0x78, 0x2a, 0x9d, 0xa5, 0xf0, 0x73, 0x22,
    0x89, 0x05, 0xf4, 0x07, 0x21, 0x52, 0xa6, 0x28,
    0x9a, 0x92, 0x69, 0x8f, 0xc5, 0xc3, 0xf5, 0xe1,
    0xde, 0xec, 0x09, 0xf2, 0xd3, 0xaf, 0x34, 0x23,
    0xaa, 0xdf, 0x7e, 0x82, 0x29, 0xc0, 0x24, 0x14,
    0x03, 0x32, 0x4e, 0x39, 0x6f, 0xc6, 0xb1, 0x9b,
    0xea, 0x72, 0x79, 0x41, 0xd8, 0x26, 0x6c, 0x5e,
    0x2c, 0xb4, 0xa2, 0x53, 0x57, 0xe2, 0x9c, 0x86,
    0x54, 0x95, 0xb6, 0x80, 0x8c, 0x36, 0x67, 0xbd,
    0x08, 0x93, 0x2f, 0x99, 0x5a, 0xf8, 0x3a, 0xd7,
    0x56, 0x84, 0xd2, 0x01, 0xf6, 0x66, 0x4d, 0x55,
    0x8b, 0x0c, 0x0b, 0x46, 0xb7, 0x3c, 0x45, 0x91,
    0xa4, 0xe3, 0x70, 0xd6, 0xfb, 0xe6, 0x10, 0xa9,
    0xc9, 0x00, 0x9e, 0xe7, 0x4f, 0x76, 0x25, 0x3f,
    0x5f, 0xa3, 0x33, 0x20, 0x02, 0xef, 0x62, 0x74,
    0xee, 0x17, 0x81, 0x42, 0x58, 0x0a, 0x4b, 0x63,
    0xe5, 0xbe, 0x6e, 0xad, 0xbf, 0x43, 0x94, 0x97,
};

```

Appendix C: the Qbox

```

WORD Qbox[256] = {
    0x1faa1887, 0x4e5e435c, 0x9165c042, 0x250e6ef4,

```


0x5957ee20,	0xd484fed3,	0xa666c502,	0x7e54e8ae,
0xd12ee9d9,	0xfc1f38d4,	0x49829b5d,	0x1b5cdf3c,
0x74864249,	0xda2e3963,	0x28f4429f,	0xc8432c35,
0x4af40325,	0x9fc0dd70,	0xd8973ded,	0x1a02dc5e,
0xcd175b42,	0xf10012bf,	0x6694d78c,	0xacaab26b,
0x4ec11b9a,	0x3f168146,	0xc0ea8ec5,	0xb38ac28f,
0x1fed5c0f,	0xaab4101c,	0xea2db082,	0x470929e1,
0xe71843de,	0x508299fc,	0xe72fbc4b,	0x2e3915dd,
0x9fa803fa,	0x9546b2de,	0x3c233342,	0x0fcee7c3,
0x24d607ef,	0x8f97ebab,	0xf37f859b,	0xcd1f2e2f,
0xc25b71da,	0x75e2269a,	0x1e39c3d1,	0xeda56b36,
0xf8c9def2,	0x46c9fc5f,	0x1827b3a3,	0x70a56ddf,
0x0d25b510,	0x000f85a7,	0xb2e82e71,	0x68cb8816,
0x8f951e2a,	0x72f5f6af,	0xe4cbc2b3,	0xd34ff55d,
0x2e6b6214,	0x220b83e3,	0xd39ea6f5,	0x6fe041af,
0x6b2f1f17,	0xad3b99ee,	0x16a65ec0,	0x757016c6,
0xba7709a4,	0xb0326e01,	0xf4b280d9,	0x4bfb1418,
0xd6aff227,	0xfd548203,	0xf56b9d96,	0x6717a8c0,
0x00d5bf6e,	0x10ee7888,	0xedfcfe64,	0x1ba193cd,
0x4b0d0184,	0x89ae4930,	0x1c014f36,	0x82a87088,
0x5ead6c2a,	0xef22c678,	0x31204de7,	0xc9c2e759,
0xd200248e,	0x303b446b,	0xb00d9fc2,	0x9914a895,
0x906cc3a1,	0x54fef170,	0x34c19155,	0xe27b8a66,
0x131b5e69,	0xc3a8623e,	0x27bdfa35,	0x97f068cc,
0xca3a6acd,	0x4b55e936,	0x86602db9,	0x51df13c1,
0x390bb16d,	0x5a80b83c,	0x22b23763,	0x39d8a911,
0x2cb6bc13,	0xbf5579d7,	0x6c5c2fa8,	0xa8f4196e,
0xbcdb5476,	0x6864a866,	0x416e16ad,	0x897fc515,
0x956feb3c,	0xf6c8a306,	0x216799d9,	0x171a9133,
0x6c2466dd,	0x75eb5dcd,	0xdf118f50,	0xe4afb226,
0x26b9cef3,	0xad36189,	0x8a7a19b1,	0xe2c73084,
0xf77ded5c,	0x8b8bc58f,	0x06dde421,	0xb41e47fb,
0xb1cc715e,	0x68c0ff99,	0x5d122f0f,	0xa4d25184,
0x097a5e6c,	0x0cbf18bc,	0xc2d7c6e0,	0x8bb7e420,
0xa11f523f,	0x35d9b8a2,	0x03da1a6b,	0x06888c02,
0x7dd1e354,	0x6bba7d79,	0x32cc7753,	0xe52d9655,
0xa9829da1,	0x301590a7,	0x9bc1c149,	0x13537f1c,
0xd3779b69,	0x2d71f2b7,	0x183c58fa,	0xacdc4418,
0x8d8c8c76,	0x2620d9f0,	0x71a80d4d,	0x7a74c473,
0x449410e9,	0xa20e4211,	0xf9c8082b,	0x0a6b334a,
0xb5f68ed2,	0x8243cc1b,	0x453c0ff3,	0x9be564a0,
0x4ff55a4f,	0x8740f8e7,	0xcc7f15f,	0xe300fe21,
0x786d37d6,	0xdfd506f1,	0x8ee00973,	0x17bbde36,
0x7a670fa8,	0x5c31ab9e,	0xd4dab618,	0xcc1f52f5,
0xe358eb4f,	0x19b9e343,	0x3a8d77dd,	0xcdb93da6,
0x140fd52d,	0x395412f8,	0x2ba63360,	0x37e53ad0,
0x80700f1c,	0x7624ed0b,	0x703dc1ec,	0xb7366795,
0xd6549d15,	0x66ce46d7,	0xd17abe76,	0xa448e0a0,

```

0x28f07c02,    0xc31249b7,    0x6e9ed6ba,    0xeaa47f78,
0xbbcffffbd,   0xc507ca84,    0xe965f4da,    0x8e9f35da,
0x6ad2aa44,    0x577452ac,    0xb5d674a7,    0x5461a46a,
0x6763152a,    0x9c12b7aa,    0x12615927,    0x7b4fb118,
0xc351758d,    0x7e81687b,    0x5f52f0b3,    0x2d4254ed,
0xd4c77271,    0x0431acab,    0xbef94aec,    0xf9e994cd,
0x9c4d9e81,    0xed623730,    0xcf8a21e8,    0x51917f0b,
0xa7a9b5d6,    0xb297adf8,    0xead30431,    0x68cac921,
0xf1b35d46,    0x7a430a36,    0x51194022,    0x9abca65e,
0x85ec70ba,    0x39aea8cc,    0x737bae8b,    0x582924d5,
0x03098a5a,    0x92396b81,    0x18de2522,    0x745c1cb8,
0xa1b8fe1d,    0x5db3c697,    0x29164f83,    0x97c16376,
0x8419224c,    0x21203b35,    0x833ac0fe,    0xd966a19a,
0xaaaf0b24f,   0x40fda998,    0xe7d52d71,    0x390896a8,
0xcee6053f,    0xd0b0d300,    0xff99cbcc,    0x065e3d40,
};

```

Appendix D: the Binary Equivalent Polynomial p_1

The Turing LFSR is equivalent to 32 parallel binary LFSRs with a characteristic polynomial (shown in binary, with the first bit being the constant term and increasing exponent):

```

100000000000000010101010101110111011101100011011000111110001101
1011101011101110011001111011011111001111100011101101101001001011
01111000101111101101011110000000000010111111101001100011010010
1011001010101111100000001001101001110100011001111110010100101011
01111101110100010101000100111010000011000110101011010000101011001
100011010011100111101000011001001100001011010110000100100011000101
111111001010010001111100110011000010111110010011000000111010111
010110100111001110111000010100000010000111000010001010001000000
10000000100000001

```

That is, $p_1(x) = 1 + x^{17} + x^{19} + x^{21} + x^{23} + x^{25} + x^{27} + x^{29} + x^{30} + \dots + x^{520} + x^{528} + x^{536} + x^{544}$. This polynomial has 267 nonzero terms.

Note: The original version of this paper gave an incorrect binary expansion of the chosen polynomial, and claimed that this polynomial had 273 nonzero terms. This error was pointed out to us by Georny Lou; many thanks.