Turing: a Fast Stream Cipher

Gregory G. Rose[1] and Philip Hawkes[1]

Qualcomm Australia, Level 3, 230 Victoria Rd, 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 [21] with a keyed mixing function reminiscent of a block cipher round. Aspects of the block mixer round have been derived from Rijndael [6], Twofish [23], tc24 [24] and SAFER++ [17].

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, and
- Able to exploit parallelism to enable fast hardware implementation.

The Turing stream cipher has a major component, the word-oriented Linear Feedback Shift Register (LFSR), which originated with the design of the SOBER family of ciphers [13, 14, 21]. Analyses of the SOBER family are found in [1–3, 11]. The efficient LFSR updating method is modelled after that of SNOW [9]. Turing combines the LFSR generator with a keyed mixing-function reminiscent of a block cipher round. The S-box used in this mixing round is partially derived from the SOBER-t32 S-box [14]. Further aspects of this mixing function have been derived from Rijndael [6], Twofish [23], tc24 [24] and SAFER++ [17].

Turing is designed to meet the needs of embedded applications 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 SOBER and SNOW, 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 [22], along with a test harness with test vectors. The reference implementation (TuringRef.c) should be viewed as the definitive description of Turing.

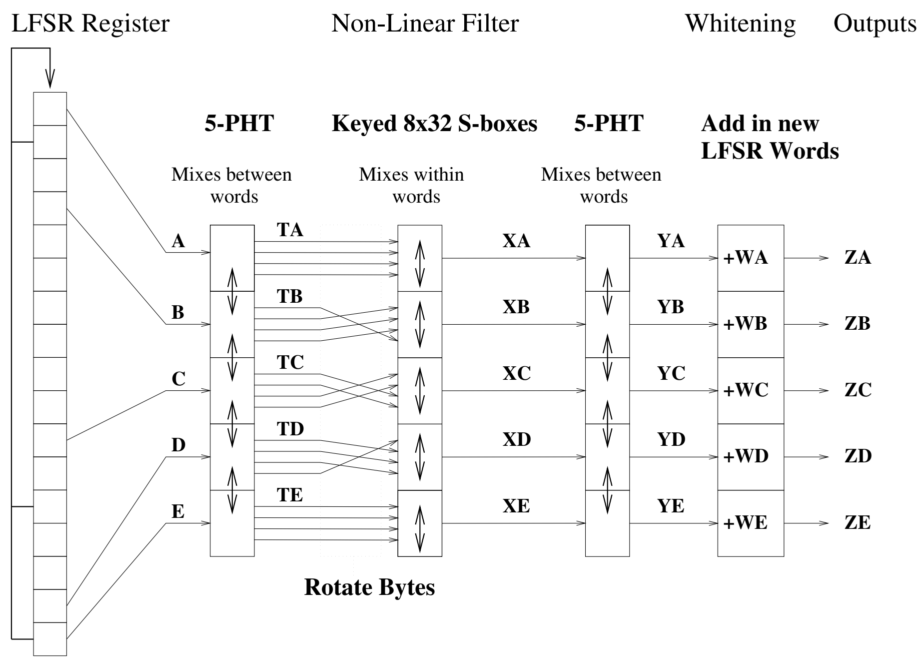

Fig. 1. Block diagram for Turing

Turing has four components: key loading, Initialisation vector (IV) loading, an LFSR, and a keyed non-linear filter (NLF). The key loading initializes the keyed S-boxes, and the IV loading initializes the LFSR. The LFSR and NLF then generate key stream in 160-bit blocks (see Figure 1). Five 32-bit words selected from the LFSR are first mixed, then passed through a highly-nonlinear, key-dependent S-box transformation, and mixed again. The resulting 5-word *nonlinear block* is combined with 5 new LFSR words to create 160 bits of keystream. The final addition of 5 LFSR words (called *whitening*) provides the output with good statistical properties, while the nonlinear block hides the linear properties of the LFSR. For each 160-bit block of key stream, the LFSR state is updated 5 times.

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.

Turing uses “big-endian” byte ordering, in which the most significant byte of a multi-byte quantity appears first in memory. For example, a 32-bit word A has bytes indexed as (A_0, A_1, A_2, A_3) where A_0 is the most significant byte.

2 LFSR of Turing

Binary Linear Feedback Shift Registers can be extremely inefficient in software on general-purpose microprocessors. LFSRs can operate over any finite field, so an LFSR 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.

The standard representation of an element A in the field $GF(2^w)$ is a w-bit word with bits $(a_{w-1}, a_{w-2}, \ldots, a_1, a_0)$, which represents the polynomial $a_{w-1}z^{w-1} + \ldots + a_1 z + a_0$. Elements can be added and multiplied: addition of elements in the field is equivalent to XOR. To multiply two elements of the field we multiply the corresponding polynomials modulo 2, and then reduce the resulting polynomial modulo a chosen irreducible polynomial of degree w.

It is also possible to represent $GF(2^w)$ using a subfield. For example, rather than representing elements of $GF(2^w)$ as degree-31 polynomials over $GF(2)$, Turing uses 8-bit bytes to represent elements of a subfield $GF(2^8)$, and 32-bit words to represent degree-3 polynomials over $GF(2^8)$. This is isomorphic to the standard representation, but not identical. The subfield $B = GF(2^8)$ of bytes is represented in Turing modulo the irreducible polynomial $z^8 + z^6 + z^3 + z^2 + 1$. Bytes represent degree-7 polynomials over $GF(2)$; the constant $\beta_0 = \mathrm{0x67}$ below represents the polynomial $z^6 + z^5 + z^2 + z + 1$ for example. The Galois finite field $W = B^4 = GF((2^8)^4)$ of words can now be represented using degree-3 polynomials where the coefficients are bytes (subfield elements of B). For example, the word 0xD02B4367 represents the polynomial $\mathrm{0xD0}y^3 + \mathrm{0x2B}y^2 + \mathrm{0x43}y + \mathrm{0x67}$. The field W can be represented by an irreducible polynomial $y^4 + \beta_3 y^3 + \beta_2 y^2 + \beta_1 y + \beta_0$. The specific coefficients β_i used in Turing are best given after describing Turing's LFSR.

An LFSR of order k over the field $GF(2^w)$ generates a stream of w-bit *LFSR words* $\{S[i]\}$ using a *register* of k memory elements `(R[0],R[1],...,R[k-1])`. The register stores the values of k successive LFSR words so after i *clocks* the register stores the values of $(S[i], S[i+1], ..., S[i+k-1])$. At each clock, the LFSR computes the next LFSR word $S[i+k]$ in the sequence using a $GF(2^w)$ recurrence relation

$$S[i+k] = \alpha_0 S[i] + \alpha_1 S[i+1] + \cdots + \alpha_{k-1} S[i+k-1] , \qquad (1)$$

and updates the register (here `new` contains the value of $S[i+k]$):

```
R[0] = R[1]; R[1] = R[2]; ...; R[15] = R[16]; R[16] = new;
```

The register now contains $(S[(i+1)], S[(i+1)+1], ..., S[(i+1)+k-1])$. The linear recurrence (1) is commonly represented by the *characteristic polynomial* $p(X) = X^k - \sum_{j=0}^{k-1} \alpha_j X^j$.

In the case of Turing, the LFSR consists of $k = 17$ words of state information with $w = 32$-bit words. The LFSR was developed in three steps. First, the characteristic polynomial of the Turing LFSR was chosen to be of the form $p(X) = X^{17} + X^{15} + X^4 + \alpha$, over $GF(2^{32})$. The exponents $\{17, 15, 4, 0\}$ were chosen because they provide good security; the use of exponents dates back to the design of SOBER-t16 and SOBER-t32 [13]. Next, the coefficient $\alpha = \text{0x00000100} \equiv \text{0x00} \cdot y^3 + \text{0x00} \cdot y^2 + \text{0x01} \cdot y + \text{0x00} = y$, was chosen because it allows an efficient software implementation: multiplication by α consists of shifting the word left by 8 bits, and adding (XOR) a pre-computed constant from a table indexed by the most significant 8 bits, as in SNOW. A portion of this table `Multab` for Turing is shown in Appendix A. In C code, the new word to be inserted in the LFSR is calculated:

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

where `^` is the XOR operation; `<<` is the left shift operation; and `>>` is the right shift operation. Finally, the irreducible polynomial representing the Galois field W was chosen to be $y^4 + \text{0xD0} \cdot y^3 + \text{0x2B} \cdot y^2 + \text{0x43} \cdot y + \text{0x67}$, since it satisfies the following constraints:

- **The LFSR must have maximum length period.** The period has a maximum length $(2^{544} - 1)$ when the field representations make $p(X)$ a primitive polynomial of degree 17 in the field W.
- **Half of the coefficients of bit-wise recurrence must be 1.** The Turing LFSR is mathematically equivalent to 32 parallel bit-wide LFSRs over $GF(2)$: each of length equivalent to the total state $17 \times 32 = 544$; each with the same recurrence relation; but different initial state [15]. Appendix D shows the polynomial $p_1(x)$, corresponding to the binary recurrence for the Turing LFSR. Requiring half of the coefficients to be 1 is ideal for maximum diffusion and strength against cryptanalysis.

The key stream is generated as follows (see Figure 1). First, the LFSR is clocked. Then the 5 values in `R[16], R[13], R[6], R[1], R[0]`, are selected as the inputs (A, B, C, D, E) (respectively) to the nonlinear filter (NLF). The NLF produces the nonlinear block (YA, YB, YC, YD, YE) from (A, B, C, D, E). The LFSR is clocked an additional three times, and the values in `R[14]`, `R[12]`, `R[8]`, `R[1]`, `R[0]` of this new state (referred to as WA, WB, WC, WD, WE) are selected for the whitening. These five words are added (modulo 2^{32}) to the corresponding nonlinear-block words to form a 160-bit key stream block (ZA, ZB, ZC, ZD, ZE). Finally, the LFSR is clocked once more before generating the next key stream block (a total of five clocks between producing outputs).

The key stream is output in the order $ZA, \ldots, ZE$; 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.

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. As shown in Figure 1, the nonlinear filter in Turing consists of:

- Selecting the 5 input words A, B, C, D, E;
- Mixing the words using a 5-word Pseudo-Hadamard Transform (5-PHT), resulting in 5 new words TA, TB, TC, TD, TE.
- Applying a 32×32 S-box construction to each of the words to form XA, XB, XC, XD, XE. Prior to applying the S-box construction, the words TB, TC and TD are rotated left by 8, 16 and 24 bits respectively, to address a potential attack described below. The S-box construction mixes the bytes within each word using four key-dependent, $8 \rightarrow 32$ nonlinear S-boxes.
- Again mixing using the 5-PHT to form the words YA, YB, YC, YD, YE of the nonlinear block .

Note that the use of variables XA, XB and so forth is only to make the explanations simple. In practise, the same variable A would be overwritten for each of TA, XA, YA, ZA, and similarly for B, C, D, E.

3.1 The "Pseudo-Hadamard Transform" (PHT)

In the cipher family of SAFER [16], Massey uses this very simple construct (called a Pseudo-Hadamard Transform) 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. The operation can further extended to mix an arbitrary number of words (often called a n-PHT). Such operations are used in the SAFER++ block cipher [17]), and the *tc24* block cipher [24]. The Turing NLF uses addition modulo 2^{32} to perform a 5-PHT:

$$\begin{bmatrix} TA \\ TB \\ TC \\ TD \\ TE \end{bmatrix} = \begin{bmatrix} 2\,1\,1\,1\,1 \\ 1\,2\,1\,1\,1 \\ 1\,1\,2\,1\,1 \\ 1\,1\,1\,2\,1 \\ 1\,1\,1\,1\,1 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix} .$$

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

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

3.2 The S-box Construction

Turing S-box construction transforms each word using four logically independent $8 \rightarrow 32$ S-boxes S_0, S_1, S_2, S_3. These $8 \rightarrow 32$ S-boxes are applied to the corresponding bytes of the input word and XORed, in a manner similar to that used in Rijndael [6]. However, unlike Rijndael, this transformation is unlikely to be invertible, as the expansion from 8 bits to 32 bits is nonlinear. These four $8 \rightarrow 32$ S-boxes are based in turn on a fixed $8 \rightarrow 8$ bit permutation denoted *Sbox* and a fixed nonlinear $8 \rightarrow 32$ bit function denoted *Qbox*, iterated with the data modified by variables derived during key setup.

The Sbox. The fixed $8 \rightarrow 8$ S-box is referred to in the rest of this document as *Sbox*[.]. It is a permutation of the input byte, has a minimum nonlinearity of 104, and is shown in Appendix B. The *Sbox* is 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, the internal permutation after keying and generating 736 bytes, forms *Sbox*. By happy coincidence, this permutation also has no fixed points (i.e. $\forall x, Sbox[x] \neq x$).

The Qbox. The *Qbox* is a fixed nonlinear $8 \rightarrow 32$-bit table, shown in Appendix c. It was developed by the Queensland University of Technology at our request [8]. 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 (same number of zeroes and ones); and the functions should be pairwise uncorrelated.

Computing the Keyed $8 \rightarrow 32$ **S-boxes**. Turing uses four keyed $8 \rightarrow 32$ S-boxes S_0, S_1, S_2, S_3. The original key is first transformed into the *mixed key* during key loading (see Section 4.1). The mixed key is accessed as bytes $K_i[j]$; 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.

Each S-box S_i ($0 \leq i \leq 3$) uses bytes from the corresponding byte positions of the scheduled key. The process is best presented in algorithmic form. The following code implements the entire S-box construction including the XOR of the four outputs of the individual $S - boxes$. The value `w` is the input word, and the integer `r` is the amount of rotation (recall that TB, TC, TD have their inputs rotated before being input to the S-box construction).

```
static WORD S(WORD w, int r)
{
    register int    i;
    BYTE        b[4];
    WORD        ws[4];
```

```
    w = ROTL(w, r);     /* cyclic rotate w to left by  r bits*/
    WORD2BYTE(w, b);    /* divide w into bytes b[0]...b[3]   */
    ws[0] = ws[1] = ws[2] = ws[3] = 0;
    for (i = 0; i < keylen; ++i) {
         /* compute b[i]=t_i and ws[i]=w_i      */
         /* B(A,i) extracts the i-th byte of A  */
    b[0] = Sbox[B(K[i],0) ^ b[0]]; ws[0] ^= ROTL(Qbox[b[0]],i+0);
    b[1] = Sbox[B(K[i],1) ^ b[1]]; ws[1] ^= ROTL(Qbox[b[1]],i+8);
    b[2] = Sbox[B(K[i],2) ^ b[2]]; ws[2] ^= ROTL(Qbox[b[2]],i+16);
    b[3] = Sbox[B(K[i],3) ^ b[3]]; ws[3] ^= ROTL(Qbox[b[3]],i+24);
    }
         /* now xor the individual S-box outputs together     */
    w  = (ws[0] & 0x00FFFFFFUL) | (b[0] << 24);    /* S_0     */
    w ^= (ws[1] & 0xFF00FFFFUL) | (b[1] << 16);    /* xor S_1 */
    w ^= (ws[2] & 0xFFFF00FFUL) | (b[2] << 8);     /* xor S_2 */
    w ^= (ws[3] & 0xFFFFFF00UL) | b[3];            /* xor S_3 */
    return w;
}
```

We shall briefly explain the process for an individual 8×32 S-box. The input byte b is combined with a key byte and passed through the fixed *Sbox*, the result is combined with another key byte, and so on, to form a temporary result:

$$t_i(x) = Sbox[K_i[N-1] \oplus Sbox[K_i[N-2] \oplus \cdots Sbox[K_i[0] \oplus x] \cdots]],$$

where $\oplus$ is the XOR operator, and N is the number of words in the key. Note that the byte function $t_i : x \to t_i(x)$ 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 $w_i(x)$; 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. Finally, the byte in position i of $w_i(x)$ is replaced with $t_i(x)$ to form the output $S_i(x)$ to ensure that the byte in position i is balanced with respect to the input.

4 Keying the Stream Cipher

For Turing, the key and Initialization Vector (IV) are presented as a byte stream, and converted to 32-bits 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 with a 32-bit key makes a good, seedable pseudorandom number generator 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.

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; the sum of the key length and IV length must be 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.

4.1 Key Loading

The original key undergoes two steps of transformation during key loading; a byte-mixing step and a word-mixing step; resulting in the mixed key.

Byte-Mixing Step: The key bytes are mixed through the fixed *Sbox* and the *Qbox*, 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 *Sbox*, using the *Qbox* in an unbalanced Feistel structure for each byte to alter the other three bytes of the word.

```
static WORD fixedS(WORD w)
{
WORD    b;
b = Sbox[B(w, 0)]; w = ((w ^      Qbox[b])     & 0x00FFFFFF) | (b << 24);
b = Sbox[B(w, 1)]; w = ((w ^ ROTL(Qbox[b],8))  & 0xFF00FFFF) | (b << 16);
b = Sbox[B(w, 2)]; w = ((w ^ ROTL(Qbox[b],16)) & 0xFFFF00FF) | (b << 8);
b = Sbox[B(w, 3)]; w = ((w ^ ROTL(Qbox[b],24)) & 0xFFFFFF00) | b;
return w;
}
```

Word-Mixing Step: An n-PHT transform forms the mixed key words. The transformation from the original key to the mixed key 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.

If the fastest implementation of Turing is desired, at this point each the four keyed S-boxes can be "pre-computed" and stored in four tables, each with 256 32-bit entries. For each S-box, the 256 outputs of the keyed S-boxes are computed and stored in a table, indexed by the corresponding input values. The combined S-box construction 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.

Note. The role of the mixing steps in the key loading is to prevent related-key attacks. For embedded applications with a key fixed in read-only memory, it is permissible to skip the key-loading stage and use a high quality cryptographic key directly as the mixed key (i.e., as if it was the output of the key-loading stage). Another alternative is to provision the mixed key (rather than the original key) directly into the hardware.

4.2 IV Loading

The Initialization Vector (IV) loading process initializes the LFSR with values derived from a non-linear mixing of the key and the IV. Let L be the length in words of the key, and I be the length in words of the IV. The LFSR register is initialized in the following manner:

- The IV words are copied into place and processed using the byte-mixing step `fixedS()` described above.
- The mixed key words are appended, without further processing.
- A single word, 0x010203LI is appended, where L and I are presumed to be hexadecimal form. 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 32×32 S-box construction (here denoted $S()$). That is, the k-th word $R[k]$ $(L+I+1 \leq k < 17)$ is set to $S(R[k-1]]+R[k-L-I-1]])$.
- Finally, once the LFSR has been filled with data, the contents are mixed with a 17-PHT. Keystream generation can now begin.

5 Performance

If sufficient random-access memory (RAM) 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. There are 4 separate implementations in the source code archive [22]:

- TuringRef.c, an unoptimized reference implementation, which also uses little RAM. It does not precompute any tables.

– TuringTab.c precomputes the keyed S-boxes when the key is set. It uses 4K bytes of RAM in addition to the 1280-byte `Multab` for 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.

Table 1 shows various performance figures. These are measured times on an IBM laptop with 900MHz mobile 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. 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 Analysis

In this section, we attempt to justify Turing's security by reference to the mechanisms by which it defeats a variety of known attacks. In this analysis we assume the attacker has direct access to the stream generator output.

Summary. 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, key-dependent 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 [7] and have detected no statistical weaknesses.

Cipher	MByte/s	cycles/B	Key (cycles)	IV setup (cycles)	tables (Bytes)	Additional RAM (Bytes)
TuringRef	6.04	149.01	477.00	4272.31	2304	68
TuringLazy	26.92	33.43	1802.70	991.80	2304	4164
TuringTab	29.94	30.06	72457.93	900.90	2304	4164
TuringFast	146.95	6.12	72417.12	882.90	2304	4164
arrsyfor	24.00	37.49	0.00	10347.42	0	258
AES enc.	33.53	26.85	239.00	0.00	20480	176

Table 1. Performance figures comparing the speed of various implementations of Turing against AES and RC4 (arrsyfor).

6.1 Period

The LFSR is clocked five times for each output block, and five is a factor of the LFSR period, the period of any cycle is $(2^{544} - 1)/5$ blocks. This corresponds to the expected period of $(2^{544} - 1)$ words.

6.2 Guess and Determine Attacks

The choice of feedback positions for the LFSR and output positions to the NLF is copied to Turing from the SOBER t-class ciphers [13] (the LFSR taps and NLF taps respectively). The taps were chosen starting with the criteria that the NLF taps 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 [10]. The combination of taps for the LFSR and NLF was then mechanically optimized against guess and determine attacks [1, 2, 12]. 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 non-invertible, and (c) requiring a large amount of output to build a large inversion table. However, the choice of output positions has proven to frustrate other attacks. It's worth noting that SOBER-t32 (and hence the underlying structure of Turing) has been extensively analyzed for the NESSIE project [19], and it seems that the structure should provide a minimum complexity exceeding the enumeration of 256-bit keys.

6.3 Analysis of the Non-linear Filter

Coppersmith et. al. have defined a fairly general model [4] for Distinguishing Attacks against nonlinear filter generators. While there could exist other attacks on the cipher, it seems that most attacks are likely to reduce to some variation on this model, so we describe our analysis in terms of this model. 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.

The attack relies on finding a highly-correlated linear relationship between the LFSR state and some function of the outputs. Courtois [5] recently described an algebraic attack on LFSR-based stream ciphers exploiting a highly-correlated Boolean functions of bits of the LFSR state and bits of the key stream. These functions are called approximations to the NLF. The approximations do not need to be linear, however the algebraic normal forms of these approximations do need to be of low order. This analysis of the NLF explains why we believe all low-order approximations to the NLF will have low correlation, thus resisting algebraic attacks.

The S-boxes. The XORing of the four outputs of the 8×32 S-boxes makes it likely that approximations require approximating the four individual 8×32

S-boxes rather than just one 8×32 S-boxes. The S-boxes in Turing are further designed to limit the correlation of low-order approximations. 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. The keying of the S-boxes provides significant protection since an attacker must either consider: (1) *average-key* correlations- expected to be negligible; or (2) *key specific* approximations- although these are difficult to find without access to the key.

A final comment on the S-boxes. Recall that the accumulated word $w_i(x)$ 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. Each byte function t_i, being a permutation, is by definition balanced. Replacing byte i of $w_i(x)$ with $t_i(x)$ forms an output of S_i that is balanced in byte i. Thus, when all the S-box outputs are XORed, the output is balanced in each output bit, and the distribution of values is uniform for each byte position. However, for a given key, the distribution of outputs from the whole 32×32 S-box construction is not uniform. When the key is fixed, the S-box construction appears to be 32×32 pseudorandom function.

The 5-PHT. The main advantage using of the 5-PHT is its speed and parallelism, however it is not as good a mixing function as could be desired. There are linear approximations between the LSBs of the inputs and output that hold with probability one and many quadratic approximations that hold with probability one.

A further undesirable characteristic is that if two of the input words A, B, C, or D are equal, they remain equal after the transformation (e.g. $A = B \Rightarrow TA = TB$). The S-boxes operate only on individual words, and so also preserve this equality. This equality is preserved in the next 5-PHT so the two words in the nonlinear block are also equal. This undesirable differential characteristic is addressed by rotating the input to the S-boxes corresponding to TB, TC, and TD.

6.4 Analysis of the Whitening

While the S-boxes do a good job of masking the linearity of the underlying LFSR, the distribution of outputs from the 32×32 S-box construction (for a given secret key) is likely to be far from uniform. Thus, over the lifetime of a key, the distribution of 160-bits outputs from the NLF will be far from uniform. Our analysis indicates that, for a given key, the NLF is a 160×160 pseudorandom function. The whitening has three effects: first, it makes the outputs uniform; second, these operations "lock in" the mixing of the last 5-PHT stage, since an attacker needs to remove the effects of these words before being able to reverse these mixing rounds; finally, by adding five new words, more than half of the register state is involved in the filter function.

Unfortunately, the whitening is linear in the LSBs of each word. The linear nature of the LFSR means that the whitening blocks satisfy bit-wise linear recurrence relations. The corresponding key stream blocks can be combined to cancel the LSBs of the whitening and get a linear relationship between the LSBs of the key stream blocks and the LSBs of the nonlinear blocks.

The choice of LFSR values used in the whitening was based on three criteria: no LFSR word should be used in the Final Addition Stage of more than one output block; the taps should be a full positive difference set; and no input to the final addition stage should be used in the NLF of more than one output block. The first criteria is the most important. If this first criterion is not satisfied, then an attacker obtains additional linear relationships between the LSBs of the key stream blocks and nonlinear blocks. When combined with the linear relationships discussed in the previous paragraph, the attacker could obtain a solvable system of equations, and Turing would be broken.

6.5 Analysis of the Key Loading and IV Loading

Key Loading. *Related Key Attacks*: Related key attacks assume that the attacker can somehow obtain key stream from keys that are closely related to that key 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 behaviour 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 whose transformation is similar to that of an unknown key appears difficult.

IV Loading. It is well known that key stream generated by a synchronous stream-cipher should not be re-used, irrespective of the security of the cipher. Turing has an integrated mechanism to support Initialization Vectors (IVs) which allows many key streams to be generated from the same shared key.

Related / Chosen Initialization Vector Attacks: 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 17-PHT transform. LFSR states derived from different IVs are less obviously related than states drawn from different segments of the same output stream.

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 technical report, Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U.K., 1998.
2. D. Bleichenbacher and S. Patel. SOBER cryptanalysis. *Fast Software Encryption, FSE'99 Lecture Notes in Computer Science, vol. 1636, L. Knudsen ed., Springer-Verlag*, pages 305–316, 1999.
3. D. Bleichenbacher, S. Patel and W. Meier. Analysis of the SOBER stream cipher. TIA Contribution TR45.AHAG/99.08.30.12.
4. D. Coppersmith, S. Halevi and C. Jutla. Cryptanalysis of Stream Ciphers using Linear Masking. *Advances in Cryptology - CRYPTO 2002, Lecture Notes in Computer Science, vol. 2442, M. Yung ed., Springer-Verlag*, pages 515–532, 2002.
5. N. T. Courtois. Higher Order Correlatoin Attacks, XL algorithm and Cryptanalysis of Toyocrypt. *Cryptology ePrint Archive, International Association for Cryptological Research (IACR), document 2002/087*, 2002. See `http://eprint.iacr.org`.
6. J. Daemen, V. Rijmen. AES Proposal: Rijndael, 2000. See `http://www.esat.kuleuven.ac.be/~rijmen/rijndael`.
7. E. Dawson, A. Clark, H. Gustafson and L. May. CRYPT-X'98, (Java Version) User Manual. *Queensland University of Technology*, 1999.
8. E. Dawson, W. Millan, L. Burnett and G. Carter. On the Design of 8*32 S-boxes. *Unpublished report, by the Information Systems Research Centre, Queensland University of Technology*, 1999.
9. P. Ekdahl and T. Johansson. SNOW - a new stream cipher, 2000. This paper is found in the NESSIE webpages: http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/snow.zip.
10. J. Dj. Golic. On Security of Nonlinear Filter Generators. *Fast Software Encryption, FSE'96 Lecture Notes in Computer Science, vol. 1039, D. Gollman ed., Springer-Verlag*, pages 27–32, 1996.
11. C. Hall and B. Schneier. An Analysis of SOBER. *Unpublished report*, 1999.
12. P. Hawkes and G. Rose. Exploiting multiples of the connection polynomial in word-oriented stream ciphers. *Advances in Cryptology, ASIACRYPT2000, Lecture Notes in Computer Science, vol. 1976, T. Okamoto ed., Springer-Verlag*, pages 302–316, 2000.
13. P. Hawkes and G. Rose. The t-class of SOBER stream ciphers, 2000. See: `http://people.qualcomm.com/ggr/QC/tclass.pdf`.
14. P. Hawkes and G. Rose. Primitive specification and supporting documentation for SOBER-t32 submission to NESSIE, 2000. See: http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sober-t32.zip.
15. T. Herlestam. On functions of Linear Shift Register Sequences. *Advances in Cryptology - EUROCRYPT '85, Lecture Notes in Computer Science, vol. 219, F. Pichler (ed.), Springer Verlag*, pages 119–129, 1986.
16. J. L. Massey. SAFER K-64: A Byte-oriented Block-Ciphering Algorithm. *Fast Software Encryption, FSE'93 Lecture Notes in Computer Science, vol. 809, Springer-Verlag*, 1993.
17. J. Massey, G. Khachatrian and M. Kuregian. Nomination of SAFER++ as Candidate Algorithm for the New European Schemes for Signatures, Integrity, and Encryption (NESSIE). September 2000. See: http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/safer++.zip.

18. A. Menezes, P. Van Oorschot and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997, Ch 6.
19. The NESSIE Project New European Schemes for Signatures, Integrity, and Encryption, 2000–2003 See: `http://www.cryptonessie.org`.
20. C. Paar. Efficieint VLSI Architectures for Bit-Parallel Computation in Galois Fields. Ph.D. Thesis, *Institute for Experimental Mathematics, University of Essen*, 1994, ISBN 3-18-332810-0.
21. G. Rose. A stream cipher based on Linear Feedback over $GF(2^8)$. *Information Security and Privacy, Third Australasian Conference, ACISP'98, Lecture Notes in Computer Science, vol. 1438, C. Boyd, E. Dawson (Eds.),Springer-Verlag*, pages 155–146, 1998.
22. G. Rose. Reference Source Code for Turing. QUALCOMM Australia, 2002. See: `http://people.qualcomm.com/ggr/QC/Turing.tgz`.
23. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson. Twofish: A 128-Bit Block Cipher. See: `http://www.counterpane.com/twofish.html`.
24. T. St. Denis. Weekend Cipher. *sci.crypt news article:* `3d4d614d_17@news.teranews.com`.

Appendix A. Portion of Multiplication Table for Turing

```
/* Multiplication table for Turing */
unsigned long Multab[256] = {
    0x00000000, 0xD02B4367, 0xED5686CE, 0x3D7DC5A9,
    0x97AC41D1, 0x478702B6, 0x7AFAC71F, 0xAAD18478,
        ...
    0x78DEE220, 0xA8F5A147, 0x958864EE, 0x45A32789,
    0xEF72A3F1, 0x3F59E096, 0x0224253F, 0xD20F6658,
};
```

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,   0xadb36189,   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,   0xcca7f15f,   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,
    0xbbcfffbd,   0xc507ca84,   0xe965f4da,   0x8e9f35da,
    0x6ad2aa44,   0x577452ac,   0xb5d674a7,   0x5461a46a,
    0x6763152a,   0x9c12b7aa,   0x12615927,   0x7b4fb118,
    0xc351758d,   0x7e81687b,   0x5f52f0b3,   0x2d4254ed,
    0xd4c77271,   0x0431acab,   0xbef94aec,   0xfee994cd,
    0x9c4d9e81,   0xed623730,   0xcf8a21e8,   0x51917f0b,
    0xa7a9b5d6,   0xb297adf8,   0xeed30431,   0x68cac921,
    0xf1b35d46,   0x7a430a36,   0x51194022,   0x9abca65e,
    0x85ec70ba,   0x39aea8cc,   0x737bae8b,   0x582924d5,
    0x03098a5a,   0x92396b81,   0x18de2522,   0x745c1cb8,
    0xa1b8fe1d,   0x5db3c697,   0x29164f83,   0x97c16376,
    0x8419224c,   0x21203b35,   0x833ac0fe,   0xd966a19a,
    0xaaf0b24f,   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):

```
1000000000000000010101010101011101110111011000110110001111100011
0110011010110011100100011110010111110011110110011110011001001100
0011110001101011011111010101110010001001111001110111101001110111
0101111001000000000101100001001011101101111110100000111010000111
1000111101011100001110000000100000010111111010001110000000101011
0111011100100011110000101111111010110011101100011111000110010100
1010111101011111111000001111011110111001000110000000101101110000 10
1000101111000111111100001010100111100001110011110111100010100000
001000011100000000001000100000001
```

That is, $p_1(x) = 1 + x^{17} + x^{19} + x^{21} + x^{23} + x^{25} + x^{27} + x^{29} + x^{30} + \cdots + x^{520} + x^{521} + x^{532} + x^{536} + x^{544}$. This polynomial has 273 nonzero terms.