Cryptanalysis of TWOPRIME

Don Coppersmith[1], David Wagner[2], Bruce Schneier[3], and John Kelsey[3]

[1] IBM Research, e-mail: copper@watson.ibm.com
[2] U.C. Berkeley, e-mail: daw@cs.berkeley.edu
[3] Counterpane Systems, e-mail: {schneier,kelsey}@counterpane.com

Abstract. Ding et al [DNRS97] propose a stream generator based on several layers. We present several attacks. First, we observe that the non-surjectivity of a linear combination step allows us to recover half the key with minimal effort. Next, we show that the various bytes are insufficiently mixed by these layers, enabling an attack similar to those on two-loop Vigenere ciphers to recover the remainder of the key. Combining these techniques lets us recover the entire TWOPRIME key. We require the generator to produce 2^{33} blocks (2^{35} bytes), or 19 hours worth of output, of which we examine about one million blocks (2^{23} bytes); the computational workload can be estimated at 2^{28} operations. Another set of attacks trades off texts for time, reducing the amount of known plaintext needed to just eight blocks (64 bytes), while needing 2^{32} time and 2^{32} space. We also show how to break two variants of TWOPRIME presented in the original paper.

1 Introduction

The TWOPRIME stream cipher [DNRS97], introduced at FSE'97, uses a 128-bit key to generate 64-bit blocks of output at each time step; these output blocks are exclusive-ORed onto the plaintext to produce ciphertext. At a high level, TWOPRIME consists of a keyed (non-bijective) cryptographic function with 64-bit inputs and 64-bit outputs, which is used in a counter-like mode to generate keystream output.

The algorithm has ten layers; the first layer is driven by a counter, and the output of each layer becomes the input to the next. We exploit weaknesses of two of the layers to produce several different attacks against the scheme. Our conclusion is that there are too few layers for cryptographic strength.

One of the main contributions of the TWOPRIME work is that the algorithm was designed so that one could prove certain statements about the security of the cipher: it has high linear complexity, good cycle length, good resistance to LSFR-synthesis attacks, and so on[1]. Nonetheless, despite the proofs of various security properties, in this paper we show how to break TWOPRIME very efficiently.

[1] Note that it is possible to prove that using any block cipher in counter mode has good linear complexity and good cycle length—at least, in the sense that [DNRS97] proved for TWOPRIME—so in retrospect these proofs are perhaps not terribly meaningful.

Our attacks fall into two natural categories. The first three attacks, discussed in Sections 4–7, recover half of the key (namely, K_2, K_3). The second category (see Sections 8–9) includes two techniques which identify the remainder of the key (K_0, K_1) once we've found K_2, K_3.

The rest of the paper is organized as follows. In Section 2 we review the TWOPRIME scheme. In Section 3 we give some preliminary remarks which will be useful in the cryptanalysis. Section 4 gives a very easy attack to recover half of the key, based on the linear map of layer 7 failing to be surjective. Section 5 shows another attack that reduces the plaintext requirements; the cost for this improvement is an increase in the amount of offline computation required. Section 6 gives a more complicated attack to recover K_2, K_3 by breaking the period of p_0p_1 into two periods of p_0 and p_1 respectively. The probabilistic analysis backing up this attack is mentioned in Section 7. In Section 8 and 9 we finish with two attacks which can be used to recover the remainder of the key in a more mundane manner. Section 10 discusses some of the computational requirements of each attack. Section 11 and 12 discuss variants of the original scheme, and some attacks on these variants. Conclusions are reserved for Section 14.

2 Description of TWOPRIME

The TWOPRIME scheme [DNRS97] uses a 128-bit key to generate 64-bit blocks of output at each time step; these output blocks are exclusive-ORed onto the plaintext to produce ciphertext. At a high level, TWOPRIME consists of a keyed function $F_K : \mathbf{Z}_{256}^8 \to \mathbf{Z}_{256}^8$ and a custom mode for using F to generate keystream output.

The mode is somewhat similar to counter mode: the input to F comes from two independent 32-bit counters. Each counter is initialized with a key-dependent value, and is stepped by adding a public constant and then reducing modulo a public 32-bit prime.

The key, consisting of 16 bytes $k_0, \ldots, k_{15}$, is divided into four 32-bit parts, named K_0, K_1, K_2 and K_3, with the convention

$$\begin{aligned} K_0 &= k_8 + k_9 2^8 + k_{10} 2^{16} + k_{11} 2^{24} \\ K_1 &= k_{12} + k_{13} 2^8 + k_{14} 2^{16} + k_{15} 2^{24} \\ K_2 &= (k_0, k_1, k_2, k_3) \\ K_3 &= (k_4, k_5, k_6, k_7). \end{aligned}$$

The algorithm has ten layers, which we will describe. The output of each layer becomes the input of the subsequent layer. With one exception, each output consists of eight bytes, and so is an element of $\mathbf{Z}_{256}^8$. The scheme is depicted graphically in Figure 1.

The first layer involves two primes, $p_0 = 2^{32} - 17$ and $p_1 = 2^{32} - 5$, and two fixed public integers a_0 and a_1. At time step t, the output of the first layer is the two 32-bit integers $r_0 = a_0 t + K_0 \pmod{p_0}$ and $r_1 = a_1 t + K_1 \pmod{p_1}$. Each is broken into four 8-bit bytes, yielding a total of eight bytes output.

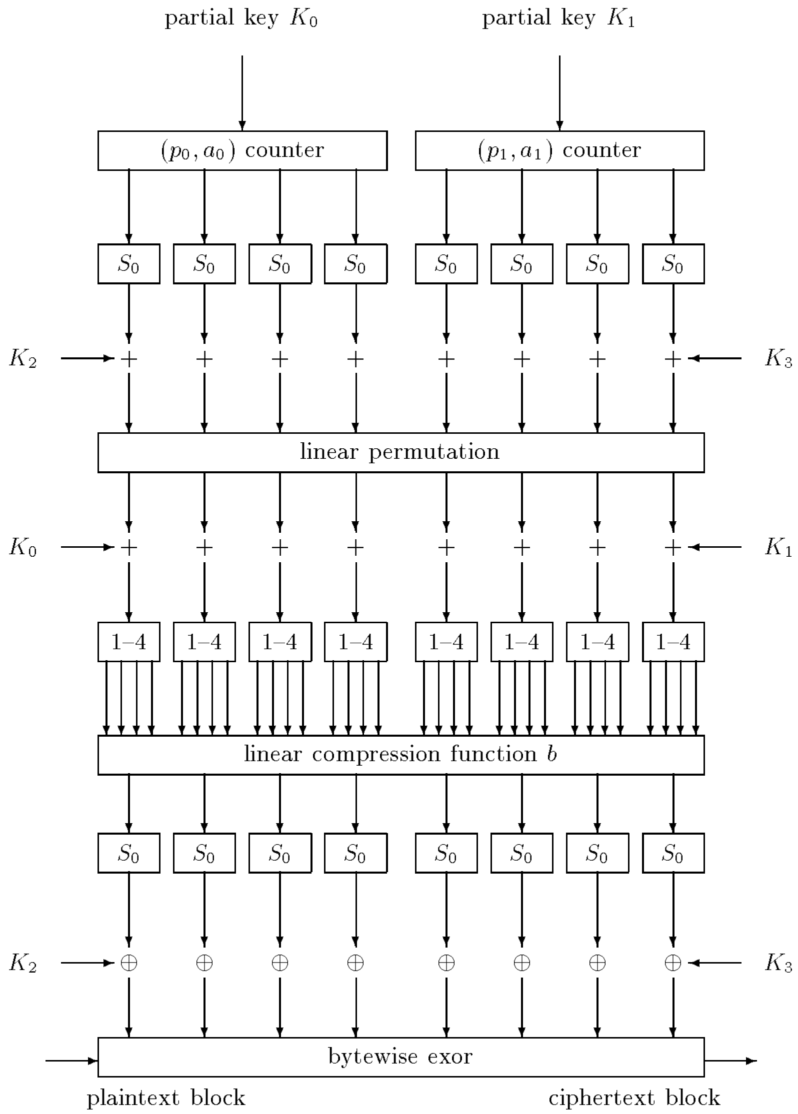

Fig. 1. Structure of the ciphering algorithm.

In the second layer, each byte x is replaced by $S_0(x) = [(x^{255} \bmod 257) \bmod 256]$. It happens that S_0 is its own inverse: $S_0(S_0(x)) = x$.

The third layer involves addition (mod 256) of the key bytes constituting K_2 and K_3.

The fourth layer is a "linear permutation": if $x_0, \ldots, x_7$ are the inputs to this layer, the outputs are

$$y_j = (\sum_{i=0}^{7} x_i) - x_j \pmod{256}.$$

This is intended to mix the bytes; however, as we shall see, it is too weak. The only interaction between the various bytes x_i is through the single byte $\sum_i x_i$ (mod 256), and when that byte is controlled, the mixing is ineffective.

The fifth layer involves addition (mod 256) of the key bytes constituting K_0 and K_1.

The sixth layer is a non-linear expansion: each byte x is expanded to the concatenation of four bytes $S_1(x), S_2(x), S_3(x), S_4(x)$, where the S_i are various nonlinear permutations on $\mathbf{Z}_{256}$. The output of this layer is 32 bytes.

The seventh layer applies a linear compression to reduce these 32 bytes back to 8 bytes; that is, a fixed public 8×32 matrix $\{b_{ij}\}$ maps $\mathbf{Z}_{256}^{32}$ to $\mathbf{Z}_{256}^{8}$. Upon input $(X_0, \ldots, X_{31})$, the linear transform b produces the output $(Y_0, \ldots, Y_7) = b(X_0, \ldots, X_{31})$ according to the equation

$$\begin{cases} Y_0 = X_0 + X_5 + X_{10} + X_{15} + X_{16} + X_{22} + X_{24} + X_{30}, \\ Y_1 = X_1 + X_6 + X_{11} + X_{12} + X_{17} + X_{23} + X_{25} + X_{31}, \\ Y_2 = X_2 + X_7 + X_8 + X_{13} + X_{18} + X_{20} + X_{26} + X_{28}, \\ Y_3 = X_3 + X_4 + X_9 + X_{14} + X_{19} + X_{21} + X_{27} + X_{29}, \\ Y_4 = X_{16} + X_{21} + X_{26} + X_{31} + X_0 + X_6 + X_8 + X_{14}, \\ Y_5 = X_{17} + X_{22} + X_{27} + X_{28} + X_5 + X_{11} + X_{13} + X_3, \\ Y_6 = X_{18} + X_{23} + X_{24} + X_{29} + X_{10} + X_{12} + X_2 + X_4, \\ Y_7 = X_{19} + X_{20} + X_{25} + X_{30} + X_{15} + X_1 + X_7 + X_9. \end{cases} \tag{1}$$

The eighth layer applies the permutation S_0 to each byte.

In the ninth layer, bytes from K_2 and K_3 are exclusive-ORed into the bytes.

The tenth round consists of exclusive-ORing these bytes (the output of the ninth round) onto the plaintext to produce the ciphertext, or (in the case of decryption) onto the ciphertext to recover plaintext.

Let us denote by $x_i^{(j)}$ $(0 \le i \le 7, 1 \le j \le 10)$ the ith byte of the output of the jth round. (For $j = 6$ we will allow $0 \le i \le 31$.) If the time step t is important we will write $x_i^{(j,t)}$. The notation $x_*^{(j)}$ will mean the whole 8-tuple of bytes $[x_i^{(j)}, 0 \le i \le 7]$.

3 Remarks on the scheme

During most of the rounds, the various bytes remain separate. During the first round, four bytes are output from one 32-bit word, and four from another. The fourth round combines bytes with a linear map, but (as has been remarked) this does a weak job of mixing them.

The seventh round combines pieces of the various bytes much more thoroughly, but only with a linear transformation. Also, the seventh round lies close to the surface, which lets us exploit the lack of diffusion in the rest of the cipher.

The designers explain that the internal structure of TWOPRIME (i.e. the function F) was chosen to resist inversion attacks (where one tries to use the output of F to work backwards). Two of our attacks succeed exactly because we can work backwards from the output of F.

In fact, we use the non-invertibility of F to our advantage in Sections 4–5. Because F is not bijective, not all intermediate values are possible. In particular, the combination of the sixth and seventh layers forms a non-surjective function, so not all 64-bit values are attainable as the output of the seventh layer. Furthermore, layers 8–10 depend only on K_2, K_3, and not on K_0, K_1. Therefore, we can isolate the effect of K_2, K_3 and attack them standing alone. Later, we can peel off layers 8–10 and use separate techniques (see Sections 8–9) to recover the remainder of the key (K_0, K_1).

4 Linear algebra

The linear recombination step (layer seven) suffers from the following regularity.

Denote by τ the 8-vector $[1,1,1,1,-1,-1,-1,-1]$. The matrix b_{ij} obeys $\sum_i \tau_i b_{ij} = 0 \pmod{256}$ for all indices j. This implies that

$$\sum_{i=0}^{7} \tau_i x_i^{(7)} = 0 \pmod{256}. \tag{2}$$

We can use this information, and a few known outputs of the stream generator, to recover the half of the key (K_2, K_3).

For each byte position i we have

$$x_i^{(7)} = S_0(x_i^{(8)}) = S_0(x_i^{(9)} \oplus k_i),$$

recalling that S_0 is its own inverse. For each i this gives a fixed mapping from $x_i^{(9)}$ to $x_i^{(7)}$, independent of time and of the other bytes.

Denote by y_{ij} the unknown quantity

$$y_{ij} = S_0(j \oplus k_i)$$

which would be the value of $x_i^{(7)}$ if $x_i^{(9)} = j$. For each block of output of the stream cipher (at time t) we obtain a linear equation relating these quantities:

$$0 = \sum_{i=0}^{7} \tau_i x_i^{(7,t)} = \sum_{i=0}^{7} \tau_i y_{i, x_i^{(9,t)}} \pmod{256}.$$

After we obtain about 2,048 blocks ($16,384$ bytes) of output, we will have 2,048 linear equations in the 2,048 unknowns y_{ij}, $0 \le i \le 7, 0 \le j \le 255$. Because of homogeneity these equations will not be independent, and for fixed i we will recover y_{ij} only up to an unknown multiplicative factor and an unknown additive shift:

$$y_{ij} = \alpha_i z_{ij} + \beta_i \pmod{256} \tag{3}$$

with z_{ij} known but α_i, β_i unknown.

But this is clearly enough information to recover the unknown key byte k_i, using a few hundred operations of trial-and-error. For each possible value for k_i, decrypt three or four values $j = x_i^{(9)}$ into $y_{ij} = S_0(j \oplus k_i)$ and check against (3). The correct k_i will be compatible with (3), and only a few others; a few more trial decryptions should rule out the false alarms.

Having determined $(k_0, \ldots, k_7) = (K_2, K_3)$, we still[2] have to find K_0 and K_1. This seems to be more expensive (and less interesting). We see a way of finding them using about 2^{32} operations and just a few known outputs of the stream cipher. See Sections 8–9.

The present attack does require about 2048 blocks (16384 bytes) of stream output. Those known plaintext requirements are not onerous, but it is possible to reduce them even further with meet-in-the-middle techniques, which we discuss next.

5 A meet-in-the-middle attack

In this attack, we take advantage of the non-surjectivity of layer seven in a different way. It is essentially a meet-in-the-middle attack, taking advantage of unattainable values at the output of the seventh layer.

Roughly speaking, we guess (K_2, K_3) and work backwards from a block of known keystream to find the output of the seventh layer, using unattainable values to rule out incorrect guesses at (K_2, K_3). This would take 2^{64} time to implement as stated; however, we have an optimization (again based on meet-in-the-middle techniques) to reduce the complexity to 2^{32}.

As before, we rely on the crucial observation (2). If we take some keystream block $x_*^{(9)}$, then inverting layers 8–9 shows that $x_i^{(7)} = S_0(x_i^{(9)} \oplus k_i)$. Plugging into (2) gives us a relation that the correct value of the key $k_0, \ldots, k_7$ must satisfy.

So the attack proceeds as follows. We define

$$g(K_2, y_0, \ldots, y_3) = \sum_{i=0}^{3} S_0(y_i \oplus k_i) \pmod{256}$$

[2] In some situations, recovering just (K_2, K_3) might conceivably suffice. After all, this gives us enough information to predict some keystream bytes: given any seven bytes from a keystream block, we can predict the eighth unknown byte with certainty by using (2). However, we can do much better. As we shall see, recovering (K_0, K_1) in a second phase requires a bit more work, but it is still feasible.

$$h(K_3, y_4, \ldots, y_7) = \sum_{i=4}^{7} S_0(y_i \oplus k_i) \pmod{256}.$$

We obtain eight known keystream blocks $x_*^{(9,j)}, 0 \leq j \leq 7$, and let

$$g'(K_2) = (g(K_2, x_0^{(9,0)}, \ldots, x_3^{(9,0)}), \ldots, g(K_2, x_0^{(9,7)}, \ldots, x_3^{(9,7)}))$$
$$h'(K_3) = (g(K_3, x_4^{(9,0)}, \ldots, x_7^{(9,0)}), \ldots, g(K_3, x_4^{(9,7)}, \ldots, x_7^{(9,7)})).$$

Note that, for the correct value of (K_2, K_3), we have $g'(K_2) = h'(K_3)$.

After all this effort to frame things in the language of meet-in-the-middle attacks, it should be clear how to recover (K_2, K_3) with standard techniques. (Here the "middle" for the meet-in-the-middle attack will be the 64-bit value $g'(K_2) = h'(K_3)$, i.e. a characteristic of the output of the seventh layer.)

First, for each guess at K_2, we compute $g'(K_2)$, and store the pair $(g'(K_2), K_2)$ in a hash table indexed on the first coordinate of the pair. After enumerating all 2^{32} possibilities for K_2, we will have constructed a hash table of size 2^{32}. Then, for each guess at K_3, we compute $h'(K_3)$ and look it up in the hash table. If we find a match $g'(K_2) = h'(K_3)$, then with high probability we will have obtained the correct values for (K_2, K_3).

We need eight keystream blocks to ensure that the test will eliminate nearly all incorrect values. One can count the number of false alarms by counting the number of solutions a, b to $g'(a) = h'(b)$. Because S_0 is highly non-linear, we are justified in expecting the functions g', h' to behave roughly like random functions of the form $\mathbf{Z}_{256}^4 \to \mathbf{Z}_{256}^8$. Combining this heuristic with the birthday paradox, we find that the probability of generating a false alarm is $1 - e^{-1} \approx 0.63$, and the expected number of false alarms is 1.

To aid the intuition, we can think of the present attack as applying a meet-in-the-middle attack *twice*, splitting the cipher first with a horizontal cut and then splitting it again with a vertical cut.

The horizontal cut is possible because layer seven fails to be surjective, and it is beneficial because layers 8–10 only depend on half of the key. (There is a slight difference, though. In a normal meet-in-the-middle attack, one computes forward part-way, backward part-way, and then meets in the middle. In our attack on TWOPRIME, because layers 6–7 fail to be surjective, we only need to compute backwards, and the forward part of the computation is substantially simplified.)

The vertical cut is made possible by the linearity of layer seven (or, more precisely, the linearity of (2)). Here the "middle" is the value $g'(K_2) = h'(K_3)$. We compute up the left half, and up the right half, and then meet in the "middle" of the output of the seventh layer. This second application of meet-in-the-middle techniques lets us isolate the effect of K_2 from that of K_3, and hence reduces the attacker's workload significantly.

In summary, we can recover (K_2, K_3) with 2^{32} offline work, 2^{32} space, and about eight blocks (64 bytes) of known keystream. As we shall see in Section 10, the computational requirements are not unreasonable.

6 Splitting the period

The previous two attacks could be avoided (in a hypothetical TWOPRIME successor) by using a different linear transformation at layer seven. So we develop here another attack against that eventuality.

This attack is similar to the attacks on two-loop Vigenere ciphers, which can be found in references [Sin68] and [Tuc70].

For an arbitrary time step t_0, let us consider the outputs at four specific time steps:

$$\begin{aligned} a &= t_0 \\ b &= t_0 + p_0 \\ c &= t_0 + p_1 \\ d &= t_0 + p_0 + p_1. \end{aligned}$$

Because the counters at layer 1 are cyclic with periods p_0 and p_1 respectively, we have

$$\begin{array}{lll} x_i^{(1,a)} = x_i^{(1,b)}, & x_i^{(1,c)} = x_i^{(1,d)}, & 0 \le i \le 3 \\ x_i^{(1,a)} = x_i^{(1,c)}, & x_i^{(1,b)} = x_i^{(1,d)}, & 4 \le i \le 7, \end{array}$$

and hence, because the actions of subsequent layers are time-invariant,

$$\begin{array}{lll} x_i^{(3,a)} = x_i^{(3,b)}, & x_i^{(3,c)} = x_i^{(3,d)}, & 0 \le i \le 3 \\ x_i^{(3,a)} = x_i^{(3,c)}, & x_i^{(3,b)} = x_i^{(3,d)}, & 4 \le i \le 7. \end{array}$$

Consider the event **E** that the following two equations both hold:

$$\begin{aligned} \sum_{i=0}^{3} x_i^{(3,a)} &= \sum_{i=0}^{3} x_i^{(3,c)} \pmod{256} \\ \sum_{i=4}^{7} x_i^{(3,a)} &= \sum_{i=4}^{7} x_i^{(3,b)} \pmod{256}. \end{aligned}$$

Each equation holds with probability about 1/256 (for randomly chosen time step t_0), and the two are independent, so that event **E** holds with probability about 1/65536. When it does hold, we have

$$\sum_{i=0}^{7} x_i^{(3,a)} = \sum_{i=0}^{7} x_i^{(3,b)} = \sum_{i=0}^{7} x_i^{(3,c)} = \sum_{i=0}^{7} x_i^{(3,d)} \pmod{256}.$$

This in turn implies that the outputs of layer 4 are well behaved:

$$\begin{array}{lll} x_i^{(4,a)} = x_i^{(4,b)}, & x_i^{(4,c)} = x_i^{(4,d)}, & 0 \le i \le 3 \\ x_i^{(4,a)} = x_i^{(4,c)}, & x_i^{(4,b)} = x_i^{(4,d)}, & 4 \le i \le 7. \end{array}$$

This can be pushed forward to give information on the outputs of layer 6:

$$\begin{array}{ll} x_i^{(6,a)} = x_i^{(6,b)}, \quad x_i^{(6,c)} = x_i^{(6,d)}, & 0 \le i \le 15 \\ x_i^{(6,a)} = x_i^{(6,c)}, \quad x_i^{(6,b)} = x_i^{(6,d)}, & 16 \le i \le 31 \\ x_i^{(6,a)} + x_i^{(6,d)} = x_i^{(6,b)} + x_i^{(6,c)}, & 0 \le i \le 31, \end{array}$$

and because layer 7 is linear (mod 256), we get

$$x_i^{(7,a)} + x_i^{(7,d)} = x_i^{(7,b)} + x_i^{(7,c)} \pmod{256}. \tag{4}$$

Suppose we know that event **E** has occurred for time step t_0, and that we have available for the output of the stream cipher $x_i^{(9,h)}$. Then from $x_i^{(7,h)} = S_0(k_i \oplus x_i^{(9,h)})$ and (4), we get a suitability test for possible values of key byte k_i. That is, for each position $0 \le i \le 7$, for each possible value of k_i, we test whether the values of $x_i^{(7,h)}$ obtained from $x_i^{(9,h)}$ using k_i would satisfy (4):

$$S_0(k_i \oplus x_i^{(9,a)}) + S_0(k_i \oplus x_i^{(9,d)}) \stackrel{?}{=} S_0(k_i \oplus x_i^{(9,b)}) + S_0(k_i \oplus x_i^{(9,c)}) \pmod{256}. \tag{5}$$

Each concatenation of possible bytes $(k_0, k_1, \ldots, k_7)$ from this step represents a possible setting of (K_2, K_3) consistent with the event **E** having occurred at this time step t_0. We will call this 8-byte setting a *putative key.*

If event **E** did occur, then the correct setting of $(k_0, k_1, \ldots, k_7)$ will be represented among these possibilities. If it did not occur, we may get several false alarms.

The difficulty is that we do not know, *a priori*, whether event **E** occurred or not. We may find that for one of the byte positions i there is no possible setting of k_i satisfying (5); in this case we know that **E** did not occur at t_0 and this case can be discarded.

Our strategy will be to try about 330,000 different values of t_0, and for each one that has at least one possible setting for each of the eight bytes k_i, record the possible values of the 8-tuple $(k_0, k_1, \ldots, k_7) = (K_2, K_3)$. The correct value should show up about five times among these putative keys, and incorrect values should show up less often. Having ascertained the correct value for (K_2, K_3), we will be able to get the keys (K_0, K_1) with less difficulty in Section 8.

7 Probabilistic analysis

For our analysis it will be useful to know the following two probability distributions.

For bytes x_a, x_b, x_c, x_d, representing $x_i^{(9,a)}, \ldots, x_i^{(9,d)}$, let $N(x_a, x_b, x_c, x_d)$ be the number of key bytes k that would satisfy (5):

$$S_0(k_i \oplus x_a) + S_0(k_i \oplus x_d) \stackrel{?}{=} S_0(k_i \oplus x_b) + S_0(k_i \oplus x_c) \pmod{256}. \tag{6}$$

We want to know the distribution $P_1(n) = \Pr(N(x_a, x_b, x_c, x_d) = n)$ when the x_h are independent random variables. We also want to know the distribution

$P_2(n) = \Pr(N(x_a, x_b, x_c, x_d) = n)$ when the x_h are known to arise from event **E**, that is, when the correct key byte k_i is known to satisfy (6). The two are related by $P_2(n) = nP_1(n)$. The experimental distributions are given in the Appendix.

The first distribution is almost Poisson with mean 1: $P_1(n) = e^{-1}/n!$, with three notable exceptions.

First, $P_1(256) \approx 2/256^2 = 2^{-15}$, because with that probability we either have ($x_a = x_b$ and $x_c = x_d$), or ($x_a = x_c$ and $x_b = x_d$), and in either case all key bytes k will work.

Second, $P_1(128) \approx (1/2)/256^2 = 2^{-17}$, and similarly $P_1(64) \approx (5/4)/256^2$, $P_1(32) \approx (13/8)/256^2$, and $P_1(16), P_1(8)$ are similarly high. This happens because of idiosyncrasies of the permutation S_0. For example, in the case $n = 128$, consider the event that $x_a \oplus x_d = x_b \oplus x_c = 11111101$ in binary, and x_a and x_b agree in the second-lowest bit. This event has probability $(1/256)^2(1/2) = 2^{-17}$. When this happens, for all 128 key bytes k disagreeing with x_a in the second-lowest bit, we have $(k \oplus x_a) + (k \oplus x_d) = 257$. Then, because $S_0(x) = x^{-1} \pmod{257}$ if $x \neq 0$, we have

$$S_0(k \oplus x_a) + S_0(k \oplus x_d) = S_0(k \oplus x_b) + S_0(k \oplus x_c) = 257$$

for each of these 128 values of k, so that $N(x_a, x_b, x_c, x_d) \geq 128$. This implies $P_1(128) \approx 2^{-17}$. Similar calculations obtain for $n = 64, 32, 16, 8$.

Third, it appears experimentally that $P_1(0)$ is a little higher than expected: 0.40 rather than 0.37; and $P_1(1)$ is a little lower. This may be related to the first two observations.

These deviations from the Poisson distribution, particularly the relative high values of $P_2(256)$ and $P_2(128)$, create a minor nuisance for our cryptanalysis.

When event **E** has happened, the distribution $P_2(n)$ is related to the number of trial key bytes k_i that would satisfy (6) in each byte position i. The number of 8-byte keys $(k_0, k_1, \ldots, k_7)$ is given by

$$\prod_{i=0}^{7} N(x_i^{(9,a)}, x_i^{(9,b)}, x_i^{(9,c)}, x_i^{(9,d)})$$

with expected value about $4.3^8 \approx 120,000$. This expected value is so high because of the unusually large values of $P_2(256)$ and $P_2(128)$.

When event **E** has not happened, the distribution $P_1(n)$ is relevant, and the expected number of 8-byte keys is 1. In fact with probability about $1 - (1 - 0.404)^8 \approx 0.984$ at least one of the values $N(x_i^{(9,a)}, x_i^{(9,b)}, x_i^{(9,c)}, x_i^{(9,d)})$ is zero, so that no 8-byte keys are valid; with the complementary probability 0.016, all are nonzero, and then the expected number of keys is $1/0.016 \approx 62$.

So with 330,000 experiments, the expected number of 8-byte putative keys is $5 \times 120,000 + (330,000 - 5) \times 1 = 930,000$. Among these, the correct key should appear five times, and should be easy to detect; incorrect keys should appear at most once, with possible exception of those differing from the correct key in only one or two bytes.

Remark: Although the mean number of putative keys is fairly small, the variance is huge; the standard deviation exceeds 10^{11}. This is because of the relatively high probability that, for a given time step and a given byte position, $N(x_a, x_b, x_c, x_d)$ is either 256 or 128; if several such bytes occur at the same time step, this time step will yield a huge number of putative keys. In this case an alternative data structure is called for. For example, if one time step has two or more such byte positions, declare that event **E** has probably occurred, and deduce putative values for the *remaining* six or fewer key bytes. Or we could simply list 4-byte putative keys K_2 and K_3 separately.

8 Splitting the period, again

Having determined K_2 and K_3 by the attack in Section 6, we also know the handful of positions where event **E** has occurred; we know several places where

$$\sum_{i=0}^{3} x_i^{(3,a)} = \sum_{i=0}^{3} x_i^{(3,c)} \pmod{256}.$$

Because of the relation between $x_i^{(3)}$ and $x_i^{(2)}$ we also have

$$\sum_{i=0}^{3} x_i^{(2,a)} = \sum_{i=0}^{3} x_i^{(2,c)} \pmod{256},$$

whence

$$\sum_{i=0}^{3} S_0(x_i^{(1,a)}) = \sum_{i=0}^{3} S_0(x_i^{(1,c)}) \pmod{256}. \tag{7}$$

By enumeration of 2^{32} possibilities, we can find all the possible values of the concatenation $(x_0^{(1,a)}, x_1^{(1,a)}, x_2^{(1,a)}, x_3^{(1,a)})$ and hence, by adding $p_1 a_0 \pmod{p_0}$, the concatenation $(x_0^{(1,c)}, x_1^{(1,c)}, x_2^{(1,c)}, x_3^{(1,c)})$, which satisfy (7). This whittles down the possible values of K_0 from a collection of 2^{32} to about $2^{32}/256^5 = 2^{12}$ possible values. Similar calculations reduce our choice of K_1 to about 2^{12} possible values. The correct values can be gotten by exhaustion.

9 Meet-in-the-middle, again

Another approach at recovering (K_0, K_1) is given here. We assume that we have previously identified (K_2, K_3) using any of the attacks from Sections 4–6. This attack requires only 2^{32} operations, 2^{24} space, and two known keystream blocks; therefore, it should be very fast.

Because of the form of the linear relation in layer 7, we find that the sum $x_0^{(7)} + x_2^{(7)} - x_4^{(7)} - x_6^{(7)} \pmod{256}$ depends only on the four bytes $x_i^{(5)}, i = 1, 3, 5, 7$. Use a meet-in-the-middle approach, requiring time $256^3 = 2^{24}$, to discover all the 2^{24} values of the 4-tuple $[x_i^{(5)}, i = 1, 3, 5, 7]$ that could lead to a given value for this sum. Similarly the sum $x_0^{(7)} + x_2^{(7)} - x_5^{(7)} - x_7^{(7)} \pmod{256}$ depends only on the four bytes $x_i^{(5)}, i = 0, 2, 4, 6$. Combine these two lists with another meet-in-the-middle attack, and in time 2^{24} we can recover the 8-tuple $x_*^{(5)}$ from any given value of the 8-tuple $x_*^{(7)}$.

Use time 2^{24} to decrypt one ciphertext back to layer 5. For each of the 2^{32} trial subkeys K_0, compute forward to $x_i^{(3)}, 0 \le i \le 3$, and backward from layer 5 to $x_i^{(4)}, 0 \le i \le 3$. See whether there is a byte sum $\sum_{i=0}^{7} x_i^{(3)}$ which would enable the linear permutation at layer 4 to map $x_i^{(3)}, 0 \le i \le 3$ to $x_i^{(4)}, 0 \le i \le 3$. We expect 256 trial subkeys K_0 to pass this test. Similarly develop 256 trial subkeys K_1. Try each of the resulting 65,536 pairs (K_0, K_1) on another ciphertext to determine the correct pair.

10 Computational requirements

The first attack should take only a few seconds to find all of K_2 and K_3, including gathering data.

The meet-in-the-middle attack recovering (K_2, K_3) (see Section 5) requires 2^{32} hash table lookups and about 2^{33} words of memory. If we keep the entire table in memory, the 2^{32} table lookups will take only 400 seconds or so (assuming 100ns access time to main memory, which is not unreasonable).

The space requirements may be more noticeable. One simple approach is to distribute the table across a cluster of 256 workstations, each with 128 MB of memory; such a cluster would take roughly 400 seconds to find (K_2, K_3). Another simple approach, if only one workstation is available, is to trade off time for memory: by splitting the table across time, one workstation can finish in $256 \times 400 \approx 10^5$ seconds (about one month), and n workstations will finish n times as fast that. This is not out of reach, and the interested reader might be able to find better ways to reduce memory needs: for example, the parallel collision search techniques of van Oorschot and Wiener [OW96] (applied to find a "golden collision") look promising.

For the attack based on identifying occurrences of event **E** (see Sections 6–8), we need the generator to run for $p_0 + p_1 \approx 2^{33}$ time steps, generating 2^{36} bytes. At the advertised speed of 1 megabyte per second, this will take about nineteen hours. We will look at only 1,000,000 message blocks (8,000,000 bytes): 330,000 at the beginning (representing a), another 330,000 in the middle (representing both b and c, because p_0 and p_1 are so close to each other), and another 330,000 at the end. For each selection (a, b, c, d) we might need to evaluate $8 \times 256 = 2048$ trial key bytes $0 \le k_i \le 255, 0 \le i \le 7$. However, realize that much of the time we will find that, for example, key byte k_1 has no possible values, so that bytes $k_2, \ldots, k_7$ need not be examined for this case. In total about 212,000,000 key bytes need to be examined.

11 TWOPRIME-1

The same paper [DNRS97] proposes a faster version TWOPRIME-1, differing from TWOPRIME only in the seventh layer; in TWOPRIME-1, this layer preserves halves. That is, the output bytes $x_i^{(7)}, 0 \leq i \leq 3$ only depend on the input bytes $x_i^{(6)}, 0 \leq i \leq 15$, and the output bytes $x_i^{(7)}, 4 \leq i \leq 7$ only depend on the input bytes $x_i^{(6)}, 16 \leq i \leq 31$. This means that the only interaction between the left and right halves of the message occurs during the "linear permutation" in the fourth layer, and there the interaction is limited to the one byte $\sum_i x_i^{(3)} \pmod{256}$. In two time steps where this sum agrees, the halves are completely separated.

So we can examine the output at time $a = t_0$ and $b = t_0 + p_0$. If $\sum_{i=4}^{7} x_i^{(3,a)} = \sum_{i=4}^{7} x_i^{(3,b)} \pmod{256}$ (i.e. the second of the two conditions for event **E**), then the left-hand half of the output of each layer is the same for a as for b:

$$x_i^{(j,a)} = x_i^{(j,b)}, \quad 0 \leq i \leq 3, \quad j \neq 6$$
$$x_i^{(6,a)} = x_i^{(6,b)}, \quad 0 \leq i \leq 15.$$

In particular the left-hand halves of the outputs will agree. By identifying eight pairs (a, b) where these output halves agree, we can deduce the value of K_0 as in the TWOPRIME case. Similar computations give us K_1.

We can then use exhaustive search to compute K_2 in about 2^{32} steps. For example, if we guess the four bytes representing $(\sum_{j=0}^{7} k_j) - k_i, 0 \leq i \leq 3$, and we know the values of K_0 and K_1, we can find the left-hand half of all layers up through layer 8. We can compare the encryptions of two unrelated time steps, say a and e, to see whether

$$x_i^{(8,a)} \oplus x_i^{(8,e)} \stackrel{?}{=} x_i^{(9,a)} \oplus x_i^{(9,e)}, \quad 0 \leq i \leq 3.$$

If not, these four bytes are wrong. But if they are equal, we can use layer 8 to deduce K_2, giving us another check on our original assumptions, and furnishing us with the correct value of K_2. The calculation of K_3 is left to the reader.

We needed to run the generator for 2^{32} messages (2^{35} bytes), or ten hours, and examine about $2 \times 8 \times 256 = 4,096$ blocks (32,768 bytes). The computational requirements of 2^{32} operations are not onerous, and the interested reader might well find more efficient methods to discover K_2.

Another approach is also available. In the first phase of this attack, we recover (K_2, K_3). The key observation is that—modelling each half of layers 6–7 as a random function—only about $1 - e^{-1}$ of the 2^{32} possible values for the left half of the output of the seventh layer will actually be attainable. Therefore, in the first phase, we guess K_2, compute up the left side of the cipher to the output of the seventh layer, and discard guesses at K_2 when they produce unattainable intermediate values. Because $(1 - e^{-1})^{50} < 2^{-32}$, we see that after about 50 blocks (400 bytes) of known plaintext, there will be just one value remaining—namely, the correct value of K_2. A similar technique recovers K_3.

Now the second phase proceeds as in Section 9. For each guess at K_0, we compute forward down the left side of the cipher to the output of layer 3 and backward to the output of layer 4, checking to see whether the two are compatible. We expect 256 values of K_0 to remain, and similarly 256 values of K_1; these remaining 2^{16} possibilities can be checked by trial encryption.

In short, this second approach breaks TWOPRIME-1 with about the same time and space complexity as the corresponding attack on TWOPRIME. We require slightly more known plaintext, but 50 blocks (400 bytes) of known plaintext should be readily available in many systems.

12 ONEPRIME

The same paper [DNRS97] proposes a scheme ONEPRIME, which differs from TWOPRIME only in the first layer: instead of two primes p_0 and p_1, we have only one prime $p = 2^{64} - 59$ and fixed multiplier a. The output of the first layer at time t is

$$(x_0^{(1)}, \ldots, x_7^{(1)}) = at + (K_0, K_1) \pmod{p}.$$

A slight modification enables our attack to run against this scheme as well. Based on the value a (which was not specified in the paper), compute values Δ_0 and Δ_1 such that in the binary representation of $a\Delta_0 \pmod{p}$, the left-most 34 bits are 0 (so that the left half is 0 and the right half represents an integer smaller than 2^{30}). Similarly in the binary representation of $a\Delta_1 \pmod{p}$, the leftmost (highest order) two bits are 0, and the rightmost 32 bits are 0. Each Δ_i should be about 2^{34} and can be computed using methods from continued fractions.

Then if we select time steps

$$\begin{aligned} a &= t_0 \\ b &= t_0 + \Delta_0 \\ c &= t_0 + \Delta_1 \\ d &= t_0 + \Delta_0 + \Delta_1 \end{aligned}$$

we will find, with probability exceeding $(3/4)^2 > 0.56$, that the left-hand halves of the outputs of layer 1 agree at times a and b, as well as at times c and d; and the right-hand halves agree at times a and c, as well as at times b and d. The rest of the attack proceeds as before.

We need the generator to run for somewhat longer, because $\Delta_0 > p_0$, and we need to examine someone more ciphertext, because our favorable conditions only occur with probability 0.56, but the attack is still feasible.

Another approach is also available. We can break ONEPRIME with meet-in-the-middle techniques. In fact, simply applying the attacks in Sections 5 and 9 immediately breaks ONEPRIME, without any modifications needed. This second approach requires eight blocks of known keystream as well as 2^{33} time and 2^{32} space.

13 Discussion

At a high level, the intuition behind some of our cryptanalysis is that we apply the meet-in-the-middle attack repeatedly, at two levels of abstraction. First, we divide the cipher horizontally between layers, and meet at the "middle"—the output of the seventh layer—at the highest level of abstraction. Second, we divide the cipher vertically into left and right halves, and meet in the "middle", where the "middle" is a characteristic of the output of the seventh layer.

Some of the techniques, e.g. Sections 6–8, do not fall cleanly into this model. We will ignore them for the moment.

Note that the vertical split can be viewed as decomposing the 64-bit function F into two parallel 32-bit functions G, H. In other words, splitting F vertically corresponds to writing $F(a, b) = (G(a), H(b))$. Of course, given such a parallel decomposition, we can apply a divide-and-conquer attack; since breaking a 32-bit function has complexity at most 2^{32}, such a decomposition lets us break F in at most $2 \cdot 2^{32}$ time.

So we conclude that F should be designed to resist parallel decomposition, and in particular there should be no parallel G, H that approximate F. This just comes down to ensuring there is plenty of diffusion, a well-known design principle for cipher design. This lack of diffusion helped make our attacks on TWOPRIME possible.

We can also analyze the horizontal split in terms of functional decomposition. In this case, we find that it corresponds to finding G, H such that $F = H \circ G$ (i.e. $F(a) = H(G(a))$). When we can find such G, H where G is non-surjective and H is bijective, then meet-in-the-middle attacks may allow the cryptanalyst to isolate the effect of G from the effect of H. In other words, the cryptanalyst can often analyze H without taking into account the effect of G (or the key bits that enter G); once H has been broken, the cryptanalyst can then peel off the effect of H (since it is bijective) and attack G alone. The result of such a divide-and-conquer attack would be that F is not much stronger than the strongest of G or H standing alone. TWOPRIME put some of its strength into G, and some into H, with the result that much of its strength was wasted. Far better would have been to concentrate all the strength in one of G or H and make the other as simple as possible, to avoid this potential danger.

Therefore, we suggest the following design principle, which seems broadly applicable to the construction of non-bijective cryptographic functions from a product of rounds. One should avoid introducing non-surjectivity in the middle of the function, because that may speed up meet-in-the-middle attacks and thus waste precious cryptographic strength.

Note that the latter design principle offers some intuitive justification for the structure of many of today's most successful non-bijective cryptographic functions (such as MD5, SHA, ...). The Davies-Meyer construction [Win84] builds F as $F(a) = G(a) \oplus a$. Here all the strength is concentrated in a bijective function G (usually built out of a block cipher); the non-surjectivity is introduced as late as possible, and as simply as possible. MD2 [Kal92] and Snefru [Mer90] also follow our suggested design principle: they too use a bijective function G at the core, and introduce non-surjectivity only at the endpoints (by adding simple redundancy to the input of G, and truncating its output).

This design principle is not novel. It has been discussed in more detail by Preneel in the context of the design of compression functions for hash functions; see [Pre93, e.g. Section 4.2].

14 Conclusions

Pulling it all together, we can identify three important attacks against the stream cipher TWOPRIME. First, we can break TWOPRIME with 2048 blocks of known keystream and 2^{32} work by using the techniques of Sections 4 and 9. Alternatively, we can get by with only 8 blocks of known keystream with repeated use of meet-in-the-middle attacks (Sections 5 and 9); the cost is that we need 2^{32} space as well as 2^{33} work. Finally, we can cryptanalyze TWOPRIME with 2^{33} blocks of known keystream and about 2^{28} operations by using the methods from Sections 6–8; this last attack uses no special features of the compression function in layer seven (other than its linearity). We see that, for a cipher with a 128-bit key, TWOPRIME is disappointingly weak.

We have pointed out weaknesses in two of the layers in TWOPRIME. Because TWOPRIME has only nine layers, each layer lies close to the surface, and any weakness is more easily exploited. The system needs more layers to have any serious cryptographic strength.

References

[DNRS97] C. Ding, V. Niemi, A. Renvall, and A. Salomaa, "TWOPRIME: A Fast Stream Ciphering Algorithm," *Fast Software Encryption, FSE'97*, Springer LNCS volume 1267, pages 88–102, 1997.

[Kal92] B.S. Kaliski, "The MD2 Message Digest Algorithm," RFC 1319, April 1992.

[Mer90] R.C. Merkle, "A Fast Software One-Way hash Function," *Journal of Cryptology*, vol 3 no 1, 1990.

[OW96] P.C. van Oorschot and M.J. Wiener, "Improving implementable meet-in-the-middle attacks by orders of magnitude," *CRYPTO'96*, pages 228-236, Springer-Verlag, 1996.

[Pre93] B. Preneel, "Design principles for dedicated hash functions," *Fast Software Encryption*, it FSE'93, Springer LNCS volume 809, pages 71–82, 1994.

[Sin68] A. Sinkov, *Elementary Cryptanalysis, A Mathematical Approach.* New York: Random House, 1968.

[Tuc70] B. Tuckerman, "A study of the Vigenere-Vernam single and multiple loop enciphering systems," IBM Research Report RC2879, 14 May 1970, Yorktown Heights NY.

[Win84] R. Winternitz, "Producing One-Way Hash Functions from DES," *Advances in Cryptology: Proceedings of Crypto 83*, Plenum Press, 1984, pp. 203–207.

A Appendix

We give here the experimental distributions of $P_1(n)$ and $P_2(n)$:

n	$e^{-1}/n!$	$P_1(n)$	$P_2(n)$
0	0.3679	0.404	0
1	0.3679	0.337	0.337
2	0.1839	0.183	0.367
3	0.0613	0.062	0.185
4	0.0153	0.017	0.070
5	0.0030	0.004	0.020
6	0.0005	0.001	0.006
7	0.0001	0.0002	0.001
8	0	0.00029	0.002
16	0	0.000028	0.0004
32	0	0.000025	0.0008
64	0	0.000019	0.0012
128	0	0.000008	0.0010
256	0	0.000031	0.0078

$$\sum nP_1(n) = 1, \quad \sum nP_2(n) \approx 4.3$$