On Fibonacci Keystream Generators

Ross Anderson

Computer Laboratory, Pembroke Street, Cambridge CB2 3QG
Email: rja14@cl.cam.ac.uk

Abstract. A number of keystream generators have been proposed which are based on Fibonacci sequences, and at least one has been fielded. They are attractive in that they can use some of the security results from the theory of shift register based keystream generators, while running much more quickly in software. However, new designs bring new risks, and we show how a system proposed at last year's workshop, the Fibonacci Shrinking Genertor (FISH), can be broken by an opponent who knows a few thousand words of keystream. We then discuss how such attacks can be avoided, and present a new algorithm, PIKE, which is based on the A5 algorithm used in GSM telephones.

1 Introduction

For many years, cryptologists have studied keystream generators based on linear feedback shift registers [1]. When implemented in hardware, such systems can use a relatively small number of gates for a given level of security; they were very popular in the days before very large scale integration, and are still used in applications such as mobile communications where low power consumption, and thus low gate count, are a priority.

However, most cryptographic algorithms are now implemented in software, and shift register generators tend to be slower in software than many alternatives. One problem is that many of them are easier to attack if their feedback polynomials are sparse (or have sparse multiples). This leads a prudent designer to specify a large number of feedback taps, or even to make the feedback depend on the key. While simple to implement in hardware, such schemes are tricky to program.

Consider for example the shrinking generator [2]. Although this is a modern algorithm, designed for efficient hardware implementation, it is scarcely faster in software than DES. The best way its implementers could find to update its key-dependent shift registers was to multiply the current state vectors by suitable binary matrices [3].

2 Generators based on Fibonacci sequences

The performance problem has led some designers of fast software encryption algorithms to abandon the shift register tradition in favour of nonlinear finite state machines [4] [5]. However, we have relatively little theory on these comparable to the cycle length and linear complexity results which can often be obtained for shift register systems, and this has led some designers to use shift register ideas to design generators based on generalised Fibonacci sequences [6] [7].

A generalised Fibonacci sequence is the sequence generated by a monic recurrence relation. More specifically, we will consider $s_i = a_{n-1}s_{i-1} + a_{n-2}s_{i-2} + ... + a_1 s_{i-n+1} + a_0 s_{i-n} \pmod{m}$, where m is a convenient power of 2.

The characteristic polynomial of such a sequence is $X^n + \sum a_i X^i$, and there are conditions on this polynomial which are necessary and sufficient for the sequence it generates to have maximal length [8]. These conditions are not quite the same as those on shift register polynomials, but they coincide for trinomials of degree greater than two. In passing we should note that the least significant bits of a Fibonacci sequence form a linear feedback shift register sequence with characteristic polynomial $X^n + \sum a'_i X^i$, where $a'_i \equiv a_i \pmod 2$.

3 The Fibonacci Shrinking Generator

We will now show how to break Siemens' Fibonacci shrinking generator (FISH), which was presented at the 1993 Cambridge Algorithms workshop [6], and is based on the shrinking generator.

It is driven by two Fibonacci sequences, which are called A and S. These start off with key material and thereafter satisfy the following recurrence relations:

$$a_i = a_{i-55} + a_{i-24} \pmod{2^{32}} \tag{1}$$

$$s_i = s_{i-52} + s_{i-19} \pmod{2^{32}} \tag{2}$$

The least significant bits of s_i are now used to shrink a_i to z_i and s_i to h_i. We will write the j-th bit of a_i as $a_{i,j}$, so that our shrinking rule is the following: if $s_{i,0} = 1$, then we append a_i to the sequence z_k, and s_i to h_k.

Next, writing $\oplus$ for bitwise xor and $\wedge$ for bitwise and, c_i is given by:

$$c_{2i} = z_{2i} \oplus (h_{2i} \wedge h_{2i+1}) \tag{3}$$

$$c_{2i+1} = z_{2i+1} \tag{4}$$

Finally, the output keystream r_i is derived from c_i by swapping the bits $c_{(2i),j}$ and $c_{(2i+1),j}$ whenever $h_{(2i+1),j} = 1$.

Now the first observation to be made about this generator is that the least significant bits of A make up a standard shrinking generator sequence; $\{z_{j,0}\}$ is just $\{a_{i,0}\}$ shrunk by $\{s_{i,0}\}$. By guessing $\{s_{i,0}\}$, we can break this sequence by brute force with an average of 2^{51} trials.

However, we can do significantly better than this. Our fish recipe will have two steps: we will first find ways to speed up the keysearch, and secondly show how to reconstruct the whole key given its least significant bits.

3.1 Sparsity and the shrinking generator

As mentioned in [2], the use of sparse shift registers in the shrinking generator can be dangerous. In fact, since A enjoys the trinomial relation (1), we will find that 1/8 of these triples $\{a_{i,0}, a_{(i+31),0}, a_{(i+55),0}\}$ will show up in $z_{i,0}$. Their separation is a random quantity, determined by S, but if we look near the likely separations, say $z_{(i-12),0} + z_{i,0} + z_{(i+15),0}$, we will expect to see this relation holding more often than random. In fact one can look at all the likely separations, $z_{(i-x),0} + z_{i,0} + z_{(i+y),0}$, where x is between (say) 12 and 19, and y is between 9 and 15.

To estimate the work factor of an attack based on this, note that every bit in $z_{i,0}$ will be the middle bit in one relation in A. Thus there is a probability of one quarter than the other two bits are in $z_{i,j}$, and about 12% that both of them will be within the ranges mentioned. If $z_{i,0} = 1$, and we find that $z_{(i-19),0}$ ... $z_{(i-12),0} = 11110111$ while $z_{(i+9),0}$ ... $z_{(i+15),0} = 111111$, we can try the six relations (0,1,1) in the knowledge that we have a 12% chance that one of them is actually a relation in A. So whenever we have enough keystream to look for a pattern as advantageous as this, we can expect to perform about 25 keysearches to recover the state of S.

Once we succeed in guessing a relation, we have a divide and conquer attack on S. If, for example, we find one at $z_{(i-12),0} + z_{i,0} + z_{(i+15),0}$, then precisely 15 of $s_{(i-31),0}$, ..., $s_{i,0}$ are one, together with precisely 12 of $s_{i,0}$, ..., $s_{(i+24),0}$; and we can derive 3 of these last 25 bits from the others using the relation (2). This reduces the attack complexity significantly (but it is still well over 2^{40} trials).

The next point is that we can cut the complexity of each key trial from $O(|A|^3)$ to $O(|A|)$ by using a trick from [9]; rather than recovering $|A|$ bits from the keystream, solving for $a_{i,0}$ using linear algebra and testing this prediction against the rest of the keystream, we simply insert bits from the keystream into $a_{i,0}$ one at a time until the recurrence relation gives us a clash. Absent an implementation, we expect that about $2|A|$ bits would suffice to detect almost all bad choices of S, it is also significant that this kind of keysearch can be implemented fairly efficiently in hardware.

The most important point, however, is that we get side information from the higher order bits of r_i about which of the possible triples $z_{(i-x),0} + z_{i,0} + z_{(i+y),0}$ represents an actual relation in the A generator. To see this, consider a series of output words r_i. For brevity, we show only the 16 rightmost (least significant) bits:

$$
\begin{aligned}
r_0 &= \texttt{0110101011010110} \\
r_1 &= \texttt{1010001101011010} \\
r_2 &= \texttt{1001011010110111} \\
r_3 &= \texttt{0011100110011011} \\
&\ldots
\end{aligned} \tag{5}
$$

Now when $h_{(2i+1),j} = 0$, $z_{(2i+1),j} = r_{(2i+1),j}$; else $z_{(2i+1),j} = r_{(2i),j}$. Thus, whenever $r_{(2i+1),j} = r_{(2i),j}$, this is equal to $z_{(2i+1),j}$. So the above table of $r_{i,j}$ gives us a partial table for $z_{i,j}$:

$$
\begin{aligned}
z_0 &= \texttt{???????????????0} \\
z_1 &= \texttt{??10?01??101??10} \\
z_2 &= \texttt{???????????????1} \\
z_3 &= \texttt{00?1????10?1??11} \\
&\ldots
\end{aligned} \tag{6}
$$

In other words, we get about half the bits of the $z_{(2i+1)}$, which is a quarter of z_i or an eighth of a_i. From here things are straightforward; given about 2^{12} words of keystream, for example, we would expect to find about one real and three false hits of the form

$$
\begin{aligned}
z_{i-x} &= \texttt{????????????0110} \\
&\ldots \\
z_i &= \texttt{????????????1011} \\
&\ldots \\
z_{i+y} &= \texttt{????????????0001}
\end{aligned} \tag{7}
$$

Finding such patterns reduces the attack complexity from about 25 key-searches to about two of them.

As we get more keystream, we will expect to find relations with x and y significantly more or less than the mean. For example, with 20 million words of keystream we would expect to find a relation which was four standard deviations better than (7), such as $x = 6$ and $y = 8$. In this case, a keysearch would involve about 2^{39} trials.

3.2 Reconstructing the rest of the key

Given $s_{i,0}$ and $a_{i,0}$, we next reconstruct the higher order bits of a_i; we know about every eighth bit of $a_{i,j}$ for each j. The tables in [10] show that with a correlation of 0.125 and a trinomial recurrence relation, a reconstruction will succeed when the length of available keystream is somewhere between 550 and 5500 bits long (and nearer the former). Of course, this recurrence relation operates modulo 2^k for $1 \leq \mathrm{k} \leq 31$ rather than over $GF(2)$, so we have to deal with carry bits as they arise; to do this, we use an algorithm like that in [11] to reconstruct first $a_{i,1}$, then $a_{i,2}$ and so on up to $a_{i,31}$.

Once we have a_i, we can get the higher order bits of s_i using those bits of r_i where $r_{(2i),j} \neq r_{(2i+1),j}$. Comparing these with our reconstructed values of z_i gives us about half of the values of $h_{(2i+1),j}$, and thus an eighth of the values of $s_{i,j}$. From this, a correlation attack can proceed as before from $s_{i,0}$ to $s_{i,1}$, $s_{i,2}$, and so on up to $s_{i,31}$.

4 An improved Fibonacci generator

The above break, together with Cain and Sherman's recent break of the Gifford cipher [12], inspires us to ask whether we can find a Fibonacci generator which is both fast in software and reasonably strong. We find some ideas in A5.

This is the algorithm used in many GSM telephones to encrypt voice traffic. It consists of three shift registers of lengths 19, 22 and 23, with sparse feedback taps, which are interlocked in the sense that a threshold function of the middle bits of each register is used to decide which registers are clocked in any given cycle (usually two of them are) [13].

The best known attack on A5 consists of guessing the state of two of the registers and then working back from the keystream to get the state of the third. There has been controversy about the work factor involved in each key trial, and at least one telecom engineer has argued that this is about 2^{12} operations giving a real attack complexity on A5 of 2^{52} rather than the 2^{40} which one might naïvely expect. As we understand that a hardware keysearch unit is under design, we expect that this debate will be settled shortly.

Nonetheless, A5 passes all the standard series randomness tests [14]; its only known weaknesses are that its registers are too short, and that it has a minimum cycle length of $\frac{4}{3}2^k$, where k is the length of the longest shift register [15] (the cycle length is not a concern in the GSM protocol, as the generator is re-keyed after each packet). A5 is also efficient; the main engineering constraint on GSM equipment is battery life, and so one may surmise that its developers sought to produce an algorithm of adequate strength, but with the smallest possible gate count. In this, they appear to have done a competent job; and this motivates us to look for a fast software algorithm which is uses the underlying ideas of A5.

Our proposal is therefore as follows. We start off with the three Fibonacci generators whose relations are:

$$a_i = a_{i-55} + a_{i-24} \pmod{2^{32}} \tag{8}$$

$$a_i = a_{i-57} + a_{i-7} \pmod{2^{32}} \tag{9}$$

$$a_i = a_{i-58} + a_{i-19} \pmod{2^{32}} \tag{10}$$

We next observe that in FISH, had the control bits been the carry bits rather than the least significant bits, then our attack would have been much harder; so we will use the carry bits as controls. If all three of them are the same, then we will step all three generators; if not, we will step those two generators whose carry bits agree. This control will be delayed eight cycles; after we update the state, we inspect the control bits and write one control nybble to a register which is shifted four bits with the next update. With some processors, it may be more convenient to use the parity bits as a control; this appears to be an acceptable variant.

The next keystream word is the xor of the least significant words of all three generators. Note that this algorithm should be slightly faster than FISH, as each keystream word will take on average 2.75 generator updates to compute rather than 3; and finally, in order to ensure that we use only a small fraction of the minimum sequence length, we specify that the generator should be rekeyed after 2^{32} words have been generated. The keying method is not part of this description, but clearly a short user supplied key can be expanded using a hash function such as SHA to provide the 700 bytes of initial state.

We call this algorithm PIKE; the pike is at the top of the food chain of our local waters, being longer, leaner and meaner than the other fish. Fishermen are of course invited to try their arm.

Acknowledgements: David Wheeler first expressed doubt about the nonlinear combining operations in FISH; Don Coppersmith pointed out the vulnerability of the sparse shrinking generator; David Wheeler pointed out the efficacy of using carry bits as controls; and Gideon Yuval remarked at the workshop that parity might be more convenient on some processors.

References

[1] RA Rueppel, *'Analysis and Design of Stream Ciphers'*, Springer Verlag Communications and Control Engineering Series (1986)

[2] D Coppersmith, H Krawczyk, Y Mansour, "The Shrinking Generator", in *Advances in Cryptology - CRYPTO '93*, Springer LNCS v **773** pp 22–39

[3] H Krawczyk, "The Shrinking Generator: some practical considerations", in *Fast Software Encryption*, Springer LNCS v **809** pp 45–46
[4] DJ Wheeler, "A Bulk Data Encryption Algorithm", in *Fast Software Encryption*, Springer LNCS v **809** pp 126–134
[5] P Rogaway, D Coppersmith, "A Software-Optimised Encryption Algorithm", in *Fast Software Encryption*, Springer LNCS v **809** pp 56–63
[6] U Blöcher, M Dichtl, "Fish: a fast software stream cipher", in *Fast Software Encryption*, Springer LNCS v 809 pp 41–44
[7] JD Golić, "Linear Cryptanalysis of Stream Ciphers", *this volume*
[8] RP Brent, "On the periods of generalised Fibonacci sequences", in *Mathematics of Computation* v **63** no 207 (July 1994) pp 389–401
[9] RJ Anderson, "Solving a Class of Stream Ciphers", in *Cryptologia* v **XIV** no 3 (July 1990) pp 285–288
[10] W Meier, O Staffelbach, "Fast Correlation Attacks on Certain Stream Ciphers", in *Journal of Cryptology* v **1** (1989) pp 159–176
[11] DJC MacKay, "A Free Energy Minimisation Framework for Inferring the State of a Shift Register given the Noisy Output Sequence", *this volume*
[12] TR Cain, AT Sherman, "How to break Gifford's Cipher", in *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (Fairfax, 1994) pp 198–209
[13] RJ Anderson, "A5 (Was: HACKING DIGITAL PHONES)", message number `<2ts9a0$95r@lyra.csx.cam.ac.uk>` posted to usenet newsgroup `sci.crypt`, 17 Jun 1994 13:43:28 GMT.
[14] M Roe, *private communication*
[15] WG Chambers, "On Random Mappings and Random Permutations", *this volume*

This article was processed using the LaTeX macro package with LLNCS style