Practical Seed-Recovery of Fast Cryptographic Pseudo-Random Number Generators

Florette Martinez
florette.martinez@lip6.fr

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Abstract. Trifork is a family of pseudo-random number generators described in 2010 by Orue *et al.* It is based on Lagged Fibonacci Generators and has been claimed as cryptographically secure. In 2017 was presented a new family of lightweight pseudo-random number generators: Arrow. These generators are based on the same techniques as Trifork and designed to be light, fast and secure, so they can allow private communication between resource-constrained devices. The authors based their choices of parameters on NIST standards on lightweight cryptography and claimed these pseudo-random number generators were of cryptographic strength.
We present practical implemented algorithms that reconstruct the internal states of the Arrow generators for different parameters given in the original article. These algorithms enable us to predict all the following outputs and recover the seed. These attacks are all based on a simple guess-and-determine approach which is efficient enough against these generators.
We also present an implemented attack on Trifork, this time using lattice-based techniques. We show it cannot have more than 64 bits of security, hence it is not cryptographically secure.

Keywords: Pseudo-random number generators, Guess-and-determine, Cryptanalysis, Lattices.

1 Introduction

Randomness is a fundamental tool in cryptography. All key generation algorithms use randomness to generate the keys and it is used in several well-known cryptographic protocols such as DSA, ECDSA, Schnorr signature scheme, etc. A pseudo-random number generator (PRNG) is an efficient deterministic algorithm that stretches a small random seed into a longer pseudo-random sequence of numbers. It is an efficient way to create pseudo-randomness to be used in cryptography protocols. A PRNG used in a cryptographic protocol needs to produce a sequence of bits indistinguishable from "truly" random bits by efficient adversaries or the whole protocol might become insecure. PRNGs of cryptographic strength exist, some of them have been approved by NIST [7].

Because of the miniaturization of components and the emergence of the Internet of Things, we face a new cryptographic challenge in which highly-constrained devices must wirelessly and securely communicate with one another. The standardized available PRNGs do not fit into these constrained devices, this is the reason why we are looking for lighter PRNGs. In [9], NIST presented several generally-desired properties that they would use to evaluate the design of future lightweight cryptographic protocols. They strongly underline the fact that the security should be of at least 112 bits.

The lagged Fibonacci generators (LFG) are a class of linear generators. A LFG is defined by four parameters: (r, s, N, m) and an initial internal state composed of r words of size N: $(x_{-r}, \ldots, x_{-1})$. At step n, the internal state of the generator is $(x_{n-r}, \ldots, x_{n-1})$. The generator computes x_n as $x_n \equiv x_{n-r} + x_{n-s} \bmod m$, outputs x_n and update its internal state to $(x_{n-r+1}, \ldots, x_n)$. These generators are light and fast, as needed for lightweight cryptography, but highly insecure. They have poor statistical properties, which make them easily distinguishable from the uniform distribution, and they are easily predictable (as we can obtain the full internal state by clocking the generator enough times).

The goal of Arrow, presented by Lopez et al. [11] was to use two of these LFGs to keep their lightweight properties by combining them in a way that would make the resulting PRNG more secure. To improve the security of these new PRNGs, the authors used two LFGs of different lengths and combined them using both modular arithmetic over $\mathbb{Z}/m\mathbb{Z}$ and modular arithmetic over $\mathbb{Z}/2\mathbb{Z}$, as combining two moduli tends to break the linearity of the operations. The sequences generated by Arrow pass successfully all the Marsaglia's Diehard randomness tests suite and the randomness tests of NIST. The statistical randomness distribution of the outputs of Arrow has been studied further in [5], by Blanco et al. in 2019.

The idea behind Arrow derives from an older family of PRNGs: Trifork. Trifork has been presented in 2010 by Orue *et al.* [12]. The generators in Trifork combine three Lagged Fibonacci Generators together, again mixing modular arithmetic over $\mathbb{Z}/m\mathbb{Z}$ and over $\mathbb{Z}/2\mathbb{Z}$. They also use a Linear Congruential Generator to initialise their large internal states. These large internal states are the main reason Trifork is not suited for lightweight cryptography. These PRNGs have a key of 192 bits and a claimed security of 192 bits.

The Linear Congruential Generators (LCG) are an other class of linear generators. A LCG is defined by three (often) public parameters a, c, m and a secret seed x_0. At step $i > 0$ the generator outputs $x_i = ax_{i-1} + c \bmod m$. These generators are well studied and generally not cryptographically secure.

Contribution. We show that Arrow, even if it has good statistical properties, is insecure. We present several practical algorithms to attack different versions of Arrow presented in the original paper, using the same choice of parameters they made for their tests. These algorithms reconstruct the full internal state of the PRNG. This allows to predict the pseudo-random stream deterministically and clock the generator backwards. For those attacks we choose a "guess-and-determine" approach: some bits of the internal state are guessed; assuming the guesses are correct, some other information is computed; a consistency check

discards bad guesses early on; then candidate internal states are computed and fully tested. Unfortunately, our attack is not general and the choice of bits to guess depends on the parameters of the underlying FLGs. This is why we need a different algorithm for each version of Arrow we want to attack. We will attack three different versions of Arrow.

	words size	key length	claimed security	attack complexity
Arrow-I	8 bits	96(128) bits	96(128) bits	38 bits
Arrow-II	16 bits	96(128) bits	96(128) bits	48 bits
Arrow-III	N bits	$32N$ bits	$32N$ bits	?

For Arrow-I and Arrow-II, the key length is 128 bits but can be shortened to 96 bits using an IV. For Arrow-III, the complexity of the attack is unclear, but the attack is practical on a laptop for $N = 8$.

We also present an attack against Trifork. The generators in Trifork have keys of length 192 bits but we show they cannot have more than 64 bits of security. Even if the two families of generators are close, the strategies to attack them greatly differ. As the internal state of Trifork is composed of several words of size 64 bits, we cannot use a "guess-and-determine" approach. As this internal state is large, it cannot be directly initialized with the key. This is why a Linear Congruential Generator is used. The LCG will be the breach we will use to attack Trifork. We will guess a third of the key (64 bits) and use lattice-based techniques to recover the rest of the seed.

Related work. Linear Congruential Generators (LCGs) have been largely studied through the years. The main attack against them was presented by Frieze *et al.* in 1984 [8]. In this attack, a Euclidean lattice related to the public parameters is built. Then outputs of the LCG are used to create a vector T1, not in the lattice but close to a vector T2 such that T2 is in the lattice and contains the seed of the generator. The lattice is reduced thanks to the LLL-algorithm -a polynomial-time reduction algorithm presented by Lenstra, Lenstra, and Lovász in 1982- and its new basis is used to solve integer linear equations to retrieve the seed of the generator. In 1985, Knuth [10] studied a variant of the LCG: the secret LCG, where the usually public parameters are now secret. This variant was attacked by Stern in 1987 [14].

Guess-and-determine (GD) techniques are mainly used to attack stream ciphers. The stream cipher SOBER was presented by Rose in 1998 [13]. In 1999, Bleichenbacher et al presented a first GD attack against SOBER-II [6] and in 2003 Baggage et al presented another GD attack against SOBER-t32 [3]. Several generators from the NESSIE competition [1] (including SOBER-t32) have been attacked with a "guess-and-determine" approach. It is also the case for the cipher stream algorithms candidate in eSTREAM [2]. You can find a quick summary of other GD uses in this survey [4], paragraph 3.10.

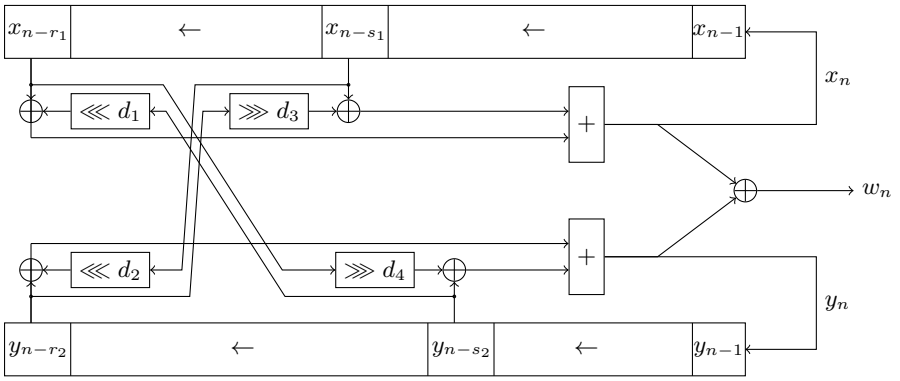


Fig. 1. Description of Arrow

2 Description of Arrow

The lagged Fibonacci generators (LFG) are a class of linear generators. A LFG is defined by four parameters: (r, s, N, m) and an initial internal state composed of r words of size N: $(x_{-r}, \ldots, x_{-1})$. At step n, the internal state of the generator is $(x_{n-r}, \ldots, x_{n-1})$. Then it computes x_n as $x_n \equiv x_{n-r} + x_{n-s} \bmod m$, outputs x_n and update its internal state to $(x_{n-r+1}, \ldots, x_n)$.

Arrow is a more elaborated architecture, its structure is described in Fig. 1. It is composed of two LFGs of respective parameters (r_1,s_1,N,m) and (r_2, s_2, N, m). The internal states of the first LFG are denoted (x_i), the internal states of the second one (y_i) and the outputs (w_i). The values $(x_i)_{-r_1 \le i \le -1}$ and $(y_i)_{-r_2 \le i \le -1}$ are the seed of this generator. The parameters r_1, r_2, s_1, s_2, N, m are public.

Instead of having $x_n = x_{n-s_1} + x_{n-r_1} \bmod m$ and $y_n = y_{n-s-2} + y_{n-r_2} \bmod m$ we scramble the two generators to obtain at step $n \ge 0$:

$$x_n = ((x_{n-r_1} \oplus (y_{n-s_2} \lll d_1)) + (x_{n-s_1} \oplus (y_{n-r_2} \ggg d_3))) \bmod m \quad (1)$$
$$y_n = ((y_{n-r_2} \oplus (x_{n-s_1} \lll d_2)) + (y_{n-s_2} \oplus (x_{n-r_1} \ggg d_4))) \bmod m \quad (2)$$

where d_1, d_2, d_3 and d_4 are four public constant satisfying $0 < d_i < N$; $\oplus$ is the bitwise exclusive-or; $\ggg$ and $\lll$ are the right-shift and left-shift operators (as defined in C, not as rotations). The output at step n is:

$$w_n = x_n \oplus y_n.$$

The security of Arrow is based on the secrecy of the internal states. If we clock r_2 times the generator, then for all $i \in \{0, \ldots, r_2 - 1\}$, we know the value $x_i \oplus y_i$ (which is equal to w_i). This is the main weakness we are going to exploit in the following attacks.

3 Attacks on Arrow

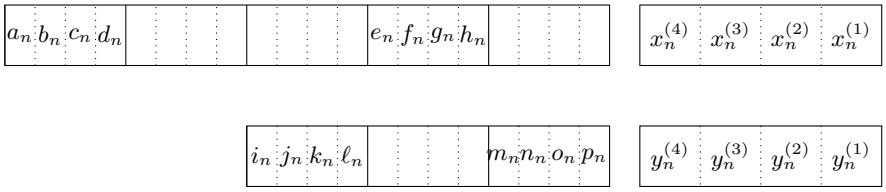
3.1 Simple Guess-and-Determine attack on Arrow-II

We present a first hardware version of Arrow (denoted Arrow-II in the introduction) with words of size $N = 16$ presented in the original paper. The set of parameters used is

N	m	r_1	s_1	r_2	s_2	$d_1 = d_2 = d_3 = d_4$
16	512	5	2	3	1	4

and the claimed security is 128 bits (96 bits if a public IV is used).

If we decide to split all the relevant words of size 16 into four sub-words of 4 bits, we can represent the internal state of this variant of Arrow as follows:



We also split the outputs w_n of size 16 into four sub outputs of 4 bits: $w_n^{(1)}$, $w_n^{(2)}$, $w_n^{(3)}$ and $w_n^{(4)}$ with $w_n^{(1)}$ being the least significant bits of w_n and $w_n^{(4)}$ the most significant bits.

The equations (1) and (2) become:

$$x_n^{(1)} = d_n + (h_n \oplus k_n) \bmod 16 \tag{3}$$

$$c_x^{(1)} = (d_n + (h_n \oplus k_n)) \operatorname{div} 16 \tag{4}$$

$$x_n^{(2)} = (c_n \oplus p_n) + (g_n \oplus j_n) + c_x^{(1)} \bmod 16 \tag{5}$$

$$c_x^{(2)} = ((c_n \oplus p_n) + (g_n \oplus j_n) + c_x^{(1)}) \operatorname{div} 16 \tag{6}$$

$$x_n^{(3)} = (b_n \oplus o_n) + (f_n \oplus i_n) + c_x^{(2)} \bmod 16 \tag{7}$$

$$c_x^{(3)} = (b_n \oplus o_n) + (f_n \oplus i_n) + c_x^{(2)} \operatorname{div} 16 \tag{8}$$

$$x_n^{(4)} = ((a_n \oplus n_n) + e_n + c_x^{(3)}) \bmod 16 \tag{9}$$

$$y_n^{(1)} = \ell_n + (c_n \oplus p_n) \bmod 16 \tag{10}$$
$$c_y^{(1)} = (\ell_n + (c_n \oplus p_n)) \operatorname{div} 16 \tag{11}$$
$$y_n^{(2)} = (h_n \oplus k_n) + (b_n \oplus o_n) + c_y^{(1)} \bmod 16 \tag{12}$$
$$c_y^{(2)} = ((h_n \oplus k_n) + (b_n \oplus o_n) + c_y^{(1)}) \operatorname{div} 16 \tag{13}$$
$$y_n^{(3)} = (g_n \oplus j_n) + (a_n \oplus n_n) + c_y^{(2)} \bmod 16 \tag{14}$$
$$c_y^{(3)} = (g_n \oplus j_n) + (a_n \oplus n_n) + c_y^{(2)} \operatorname{div} 16 \tag{15}$$
$$y_n^{(4)} = ((f_n \oplus i_n) + m_n + c_y^{(3)}) \bmod 16 \tag{16}$$

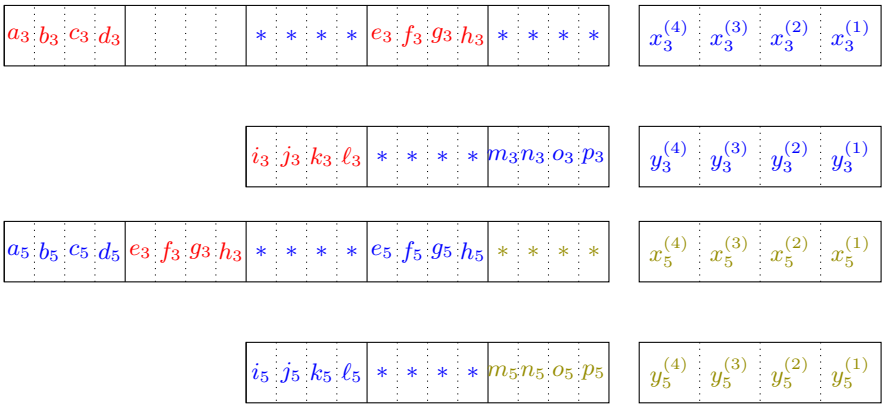
$$x_n^{(1)} \oplus y_n^{(1)} = w_n^{(1)} \tag{17}$$
$$x_n^{(2)} \oplus y_n^{(2)} = w_n^{(2)} \tag{18}$$
$$x_n^{(3)} \oplus y_n^{(3)} = w_n^{(3)} \tag{19}$$
$$x_n^{(4)} \oplus y_n^{(4)} = w_n^{(4)} \tag{20}$$

were "div" denotes the *integer division*. The $c_x^{(i)}$ and $c_y^{(i)}$ are the carries we must work with. Their value is either 0 or 1. The (w_i) are *known* as they are the outputs.

Our attack will be based on a classical "guess-and-determine" approach. The guessed bits will appear in red, the derived bits at the first step in blue, and the derived bits at the second step in olive. In this case, the attack is very simple: we start by clocking 3 times our generator.

Step 1 We guess a_3, b_3, c_3, d_3, e_3, f_3, g_3, h_3, i_3, j_3, k_3, ℓ_3 (hence 48 bits). With d_3, h_3 and k_3 we compute $x_3^{(1)}$ and $c_x^{(1)}$(eq. 3 and 4). Then we compute $y_3^{(1)}$ with $x_3^{(1)}$ and $w_3^{(1)}$ (eq. 17) and retrieve p_3 as we know ℓ_3 and c_3 (eq. 10). The knowledge of c_3 allows us to compute $x_3^{(2)}$ (eq. 5), recover $y_3^{(2)}$(eq. 18) and then o_3 (eq. 12). With o_3 we can compute $x_3^{(3)}$ (eq. 7), recover $y_3^{(3)}$ (eq. 19) and then n_3 (eq. 14). And finally, with n_3 we can compute $x_3^{(4)}$ (eq. 9) and recover $y_3^{(4)}$ (eq. 20) as well as m_3 (eq. 16). As we know w_0, w_1, w_2, we can fill up the internal states above i_3, j_3, k_3, ℓ_3 and m_3, n_3, o_3, p_3 and under e_3, f_3, g_3, h_3 (eq 17, 18, 19 and 20).

Step 2 We clock the generator twice. As explained above, we have derived a_5, b_5, c_5, d_5 from i_3, j_3, k_3, ℓ_3 and w_0. The values e_5, f_5, g_5, h_5 are $x_3^{(4)}$, $x_3^{(3)}$, $x_3^{(2)}$, $x_3^{(1)}$ and i_5, j_5, k_5, ℓ_5 are m_3, n_3, o_3, p_3. We remark that we are in a similar situation as step 1, hence we use the same equations to derive m_5, n_5, o_5, p_5 as well as $x_5^{(1)}$, $x_5^{(2)}$, $x_5^{(3)}$, $x_5^{(4)}$, $y_5^{(1)}$, $y_5^{(2)}$, $y_5^{(3)}$ and $y_5^{(4)}$.
The values above m_5, n_5, o_5, p_5 can be computed thanks to w_4.
At this point, we know the full internal state of the generator.



Step 3 We compute the five following outputs using the internal states we have and we compare them with the true outputs given by the generator. If they are equal it means we have recovered the full internal state of the generator with overwhelming probability. We notice that the generator is easily invertible, hence we can recover the seed.

This particular version of Arrow was supposed to have between 96 and 128 bits of security (depending on whether or not an IV was used) and with this attack, we show it cannot have more than 48 bits of security which is far from the 112 bits of security recommended by NIST for lightweight cryptography. This attack had been implemented in C but is not practical on a standard laptop: a Dell Latitude 7400, running on Ubuntu 18.04 (the same laptop will be used for the rest of this paper). If we only test a hundred sets of guesses, the algorithm runs in 0.000144s. To retrieve the full internal state of the generator, the algorithm should run for approximately 12 years.

3.2 Longer Guess-and-Determine attack on Arrow-I

Arrow-I is another hardware version of Arrow presented in the original paper, this time with words of size $N = 8$. The set of parameters used is

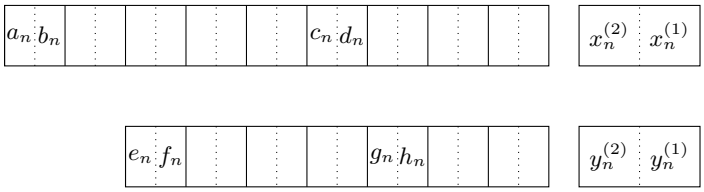
N	m	r_1	s_1	r_2	s_2	$d_1 = d_2 = d_3 = d_4$
8	256	9	4	7	3	4

and the claimed security is 128 bits (96 bits if a public IV is used).

If we decide to split all the relevant words of 8 bits into four sub-words of 4 bits, we can represent the internal state of this variant of Arrow as follows:

We also split the outputs w_n of 8 bits in two sub words of 4 bits: $w_n^{(1)}$ and $w_n^{(2)}$, with $w_n^{(1)}$ being the least significant bits of w_n and $w_n^{(2)}$ the most significant bits.

The equations (1) and (2) become:



$$x_n^{(1)} = b_n + (e_n \oplus d_n) \bmod 2^{N/2} \tag{21}$$
$$c_x = (b_n + (e_n \oplus d_n)) \operatorname{div} 2^{N/2} \tag{22}$$
$$y_n^{(1)} = f_n + (a_n \oplus h_n) \bmod 2^{N/2} \tag{23}$$
$$c_y = (f_n + (a_n \oplus h_n)) \operatorname{div} 2^{N/2} \tag{24}$$
$$x_n^{(2)} = (c_n + (a_n \oplus h_n) + c_x) \bmod 2^{N/2} \tag{25}$$
$$y_n^{(2)} = (g_n + (e_n \oplus d_n) + c_y) \bmod 2^{N/2} \tag{26}$$

We start the attack by clocking the generator seven times. Then, for every $n \geq 7$, $e_n, f_n = y_{n-7}^{(2)}, y_{n-7}^{(1)}$, $c_n, d_n = x_{n-4}^{(2)}, x_{n-4}^{(1)}$ and $g_n, h_n = y_{n-3}^{(2)}, y_{n-3}^{(1)}$. If we denote $\bar{e_i}, \bar{f_i}$ the values above e_i, f_i, we see that we can easily derive them from e_i, f_i and w_{i-7}. We also denote $\bar{g_i}, \bar{h_i}$ the values above g_i, h_i and $\bar{c_i}, \bar{d_i}$ the values under c_i, d_i

Step 0: guess $b_7, g_7, (e_7 \oplus d_7), (a_7 \oplus h_7)$
determine $\rightarrow (x_7^{(1)}, y_7^{(1)}, f_7, y_7^{(2)}, x_7^{(2)}, c_7)$

Step 1: $b_9 = \bar{f_7}$
guess $g_9, (e_9 \oplus d_9), (a_9 \oplus h_9)$
determine $\rightarrow (x_9^{(1)}, y_9^{(1)}, f_9, y_9^{(2)}, x_9^{(2)}, c_9)$

Step 2: $b_{11} = \bar{f_9}, c_{11} = x_7^{(2)}, d_{11} = x_7^{(1)}, e_{11} = g_7$
guess f_{11}
determine $\rightarrow (x_{11}^{(1)}, y_{11}^{(1)}, x_{11}^{(2)}, y_{11}^{(2)}, g_{11})$

Step 3: $a_{12} = c_7, c_{12} = \bar{g_{11}}, e_{12} = \bar{c_9}, g_{12} = y_9^{(2)}, h_{12} = y_9^{(1)}$
guess b_{12}, c_x, c_y
determine $\rightarrow (x_{12}^{(2)}, y_{12}^{(2)}, d_{12}, x_{12}^{(1)}, y_{12}^{(1)}, f_{12})$

Step 4: $a_{15} = \bar{g_9}, c_{15} = x_{11}^{(2)}, d_{15} = x_{11}^{(1)}, e_{15} = g_{11}, g_{15} = y_{12}^{(2)}, h_{12} = y_{12}^{(1)}$
determine $\rightarrow (y_{15}^{(1)}, x_{15}^{(1)}, b_{15}, x_{15}^{(2)}, y_{15}^{(2)})$

Step 5: $a_{16} = x_7^{(2)}, b_{16} = x_7^{(1)}, c_{16} = x_{12}^{(2)}, d_{16} = x_{12}^{(1)}, e_{16} = y_9^{(2)}, f_{16} = y_9^{(1)}$
determine $\rightarrow (x_{16}^{(1)}, y_{16}^{(1)}, h_{16}, x_{16}^{(2)}, y_{16}^{(2)}, g_{16})$

Step 6: $a_{18} = x_9^{(2)}, b_{18} = x_9^{(1)}, e_{18} = y_{11}^{(2)}, f_{18} = y_{11}^{(1)}, g_{18} = y_{15}^{(2)}; h_{18} = y_{15}^{(1)}$
determine $\rightarrow (y_{18}^{(1)}, x_{18}^{(1)}, d_{18}, y_{18}^{(2)}, x_{18}^{(2)}, c_{18})$

Step 7: $a_{20} = x_{11}^{(2)}, b_{20} = x_{11}^{(1)}, c_{20} = x_{16}^{(2)}, d_{20} = x_{16}^{(1)}, e_{20} = g_16, f_{20} = h_16$
determine $\rightarrow (x_{20}^{(1)}, y_{20}^{(1)}, h_{20}, x_{20}^{(2)}, y_{20}^{(2)}, g_{20})$

Step 8: $a_{21} = x_{12}^{(2)}, b_{21} = x_{12}^{(1)}, c_{21} = \bar{g_{20}}, d_{21} = \bar{h_{20}}, e_{21} = y_{12}^{(2)}, f_{21} = y_{12}^{(1)}, g_{21} = y_{18}^{(2)}, h_{21} = y_{18}^{(1)}$
determine $\rightarrow (x_{21}^{(1)}, y_{21}^{(1)}, x_{21}^{(2)}, y_{21}^{(2)})$

Step 9: $a_{22} = \bar{g_{16}}, b_{22} = \bar{h_{16}}, c_{22} = x_{18}^{(2)}, d_{22} = x_{18}^{(1)}, e_{22} = y_{15}^{(2)}, f_{22} = y_{15}^{(1)}$
determine $\rightarrow (x_{22}^{(1)}, y_{22}^{(1)}, h_{22}, x_{22}^{(2)}, y_{22}^{(2)}, g_{22})$

At the end of Step 9, we have derived from our guesses the whole internal state of the generator. We use these values to compute the five following outputs and compare them to the five "true" outputs given by the original generator to know if our guesses were correct or not with overwhelming probability. As we guess 16 bits in Step 1, 12 bits in Step 2, 4 bits in Step 3, and 6 bits in Step 4, our time complexity will be approximately (2^{38}). We recall that the security of this generator was supposed to be of at least 96 bits. This attack had been implemented in C and the expected running time is 4 hours on a standard laptop (without optimization).

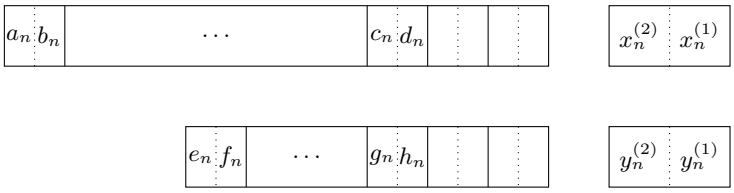
3.3 An attack against Arrow-III, the software version of Arrow

The software version of Arrow with words of size N is using the following set of parameters

N	m	r_1	s_1	r_2	s_2	$d_1 = d_2 = d_3 = d_4$
N	2^N	31	3	17	3	$N/2$

with $N = 8$ or $N = 32$.

If we decide to split all the relevant words of N bits into two sub-words of N/2 bits, we can represent the internal state of this variant of Arrow as follows:



We obtain the same equations as in the previous case.
This version of Arrow has two specificities:

- The values c_i, d_i are above g_i, h_i. Hence if the generator has been clocked enough times and if we know g_i and h_i, then we know c_i and d_i.
- The parameters satisfy $r_1 - r_2 = s_2 - r_2$. Then, if we know e_i, f_i, c_i, d_i we will know $a_{i+14}, b_{i+14}, e_{i+14}, f_{i+14}$. The two lagged Fibonacci generator used in this version of Arrows are more or less synchronized. Which is something that should have been avoided.

Because of that, in our attack we will only face three cases:

Case gh We know a_i, b_i, e_i, f_i, we guess g_i, h_i and derive c_i, d_i, x_i, y_i with the help of w_{i-3} and w_i. We compare $x_i^{(2)} \oplus y_i^{(2)}$ to $w_i^{(2)}$.

Case a We know e_i, f_i, g_i, h_i, we guess a_i and derive x_i, y_i with the help of w_i. We compare $x_i^{(2)} \oplus y_i^{(2)}$ to $w_i^{(2)}$.

Case 0 We know all the relevant values, we derive x_i, y_i from them and compare $x_i \oplus y_i$ to the output w_i.

We start by clocking the generator 17 times to know all the xor between x_i and y_i for i in $\{0, \ldots, 16\}$.

Step 0: guess $a_{17}, e_{17}, f_{17}, g_{17}, h_{17}$
determine $\rightarrow (c_{17}, d_{17}, x_{17}^{(1)}, x_{17}^{(2)}, y_{17}^{(1)}, y_{17}^{(2)})$
assert $x_{17}^{(2)} \oplus y_{17}^{(2)} = w_{17}^{(2)}$

Step 1 (case gh): $a_{31} = e_{17}, b_{31} = f_{17}, e_{31} = g_{17}, f_{31} = h_{17}$
guess g_{31}, h_{31}
determine $\rightarrow (c_{31}, d_{31}, x_{31}, y_{31})$
assert $x_{31}^{(2)} \oplus y_{31}^{(2)} = w_{31}^{(2)}$

Step 2 (case a): $c_{34} = x_{31}^{(2)}, d_{34} = x_{31}^{(1)}, e_{34} = y_{17}^{(2)}, f_{34} = y_{17}^{(1)}, g_{34} = y_{31}^{(2)}, h_{34} = y_{31}^{(1)}$
guess a_{34}
determine $\rightarrow (x_{34}, y_{34})$
assert $x_{34}^{(2)} \oplus y_{34}^{(2)} = w_{34}^{(2)}$

Step 3 (case gh): $a_{45} = c_{17}, b_{45} = d_{17}, e_{45} = g_{31}, f_{45} = h_{31}$
guess g_{45}, h_{45}
determine $\rightarrow (c_{45}, d_{45}, x_{45}, y_{45})$
assert $x_{45}^{(2)} \oplus y_{45}^{(2)} = w_{45}^{(2)}$

Step 4 (case 0): $a_{48} = x_{17}^{(2)}, b_{48} = x_{17}^{(1)}, c_{48} = x_{45}^{(2)}, d_{48} = x_{45}^{(1)}, e_{48} = y_{31}^{(2)}, f_{48} = y_{31}^{(1)}, g_{48} = y_{45}^{(2)}, h_{48} = y_{45}^{(1)}$
determine $\rightarrow (x_{48}, y_{48})$
assert $x_{48} \oplus y_{48} = w_{48}$

There are 2^{5N} possibilities for the set of values $\{a_{17}, e_{17}, f_{17}, g_{17}, h_{17}, g_{31}, h_{31}, a_{34}, g_{45}, h_{45}\}$. On average, only 2^{2N} possibilities pass the fourth step.

Step 5 (case a): $c_{51} = x_{48}^{(2)}, d_{51} = x_{48}^{(1)}, e_{51} = y_{34}^{(2)}, f_{51} = y_{34}^{(1)}, g_{51} = y_{48}^{(2)}, h_{51} = y_{48}^{(1)}$
guess a_{51}
determine $\rightarrow (x_{51}, y_{51})$
assert $x_{51}^{(2)} \oplus y_{51}^{(2)} = w_{51}^{(2)}$

Step 6 (case gh): $a_{59} = c_{31}, b_{59} = d_{31}, e_{59} = g_{45}, f_{59} = h_{45}$
guess g_{59}, h_{59}
determine $\rightarrow (c_{59}, d_{59}, x_{59}, y_{59})$
assert $x_{59}^{(2)} \oplus y_{59}^{(2)} = w_{59}^{(2)}$

Step 7 (case 0): $a_{62} = x_{31}^{(2)}, b_{62} = x_{31}^{(1)}, c_{62} = x_{59}^{(2)}, d_{62} = x_{59}^{(1)}, e_{62} = y_{45}^{(2)}, f_{62} = y_{45}^{(1)}, g_{62} = y_{59}^{(2)}, h_{62} = y_{59}^{(1)}$
determine $\rightarrow (x_{62}, y_{62})$
assert $x_{62} \oplus y_{62} = w_{62}$

Step 8 (case 0): $a_{65} = x^{(2)}_{34}, b_{65} = x^{(1)}_{34}, c_{65} = x^{(2)}_{62}, d_{65} = x^{(1)}_{62}, e_{65} = y^{(2)}_{48}, f_{65} = y^{(1)}_{48}, g_{65} = y^{(2)}_{62}, h_{65} = y^{(1)}_{62}$
determine $\rightarrow (x_{65}, y_{65})$
assert $x_{65} \oplus y_{65} = w_{65}$

We add $\{a_{51}, g_{59}, h_{59}\}$ to our set of guessed values ($2^{13N/2}$ possibilities). On average only $2^{N/2}$ pass the eighth step.

Step 9 (case a): $c_{68} = x^{(2)}_{65}, d_{68} = x^{(1)}_{65}, e_{68} = y^{(2)}_{51}, f_{68} = y^{(1)}_{51}, g_{68} = y^{(2)}_{65}, h_{68} = y^{(1)}_{65}$
guess a_{68}
determine $\rightarrow (x_{68}, y_{68})$
assert $x^{(2)}_{68} \oplus y^{(2)}_{68} = w^{(2)}_{68}$

Step 10 (case gh): $a_{73} = c_{45}, b_{73} = d_{45}, e_{73} = g_{59}, f_{73} = h_{59}$
guess g_{73}, h_{73}
determine $\rightarrow (c_{73}, d_{73}, x_{73}, y_{73})$
assert $x^{(2)}_{73} \oplus y^{(2)}_{73} = w^{(2)}_{73}$

Step 11 (case 0): $a_{76} = x^{(2)}_{45}, b_{76} = x^{(1)}_{45}, c_{76} = x^{(2)}_{73}, d_{76} = x^{(1)}_{73}, e_{76} = y^{(2)}_{59}, f_{76} = y^{(1)}_{59}, g_{76} = y^{(2)}_{73}, h_{76} = y^{(1)}_{73}$
determine $\rightarrow (x_{76}, y_{76})$
assert $x_{76} \oplus y_{76} = w_{76}$

Step 12 (case 0): $a_{79} = x^{(2)}_{48}, b_{79} = x^{(1)}_{48}, c_{79} = x^{(2)}_{76}, d_{79} = x^{(1)}_{76}, e_{79} = y^{(2)}_{62}, f_{79} = y^{(1)}_{62}, g_{79} = y^{(2)}_{76}, h_{79} = y^{(1)}_{76}$
determine $\rightarrow (x_{79}, y_{79})$
assert $x_{79} \oplus y_{79} = w_{79}$

Step 13 (case 0): $a_{82} = x^{(2)}_{51}, b_{82} = x^{(1)}_{41}, c_{82} = x^{(2)}_{79}, d_{82} = x^{(1)}_{79}, e_{82} = y^{(2)}_{65}, f_{82} = y^{(1)}_{65}, g_{82} = y^{(2)}_{79}, h_{82} = y^{(1)}_{79}$
determine $\rightarrow (x_{82}, y_{82})$
assert $x_{82} \oplus y_{82} = w_{82}$

We add $\{a_{68}, g_{73}, h_{73}\}$ to our set of guessed values (2^{8N} possibilities). On average only 2^2 pass the thirteenth step.

We keep repeating these three steps until we reach $n \simeq 250$. It takes another 110 steps to go there. At this point, we will have derived the full internal state of the generator and only one guess would have passed all the filters with overwhelming probability. This attack had been fully implemented in C. For $N = 8$ the attack is practical as it runs in 20 seconds over 8 threads on a standard laptop: a Dell Latitude 7400, running on Ubuntu 18.04.

4 Description of Trifork

Trifork's structure is described in Fig. 2.

A Trifork generator is going to use three LFGs of respective parameters (r_1, s_1, N, m), (r_2, s_2, N, m) and (r_3, s_3, N, m). The internal states of the first LFG are denoted (X_i), the internal states of the second one (Y_i), the ones of the third (Z_i) and the outputs (w_i).

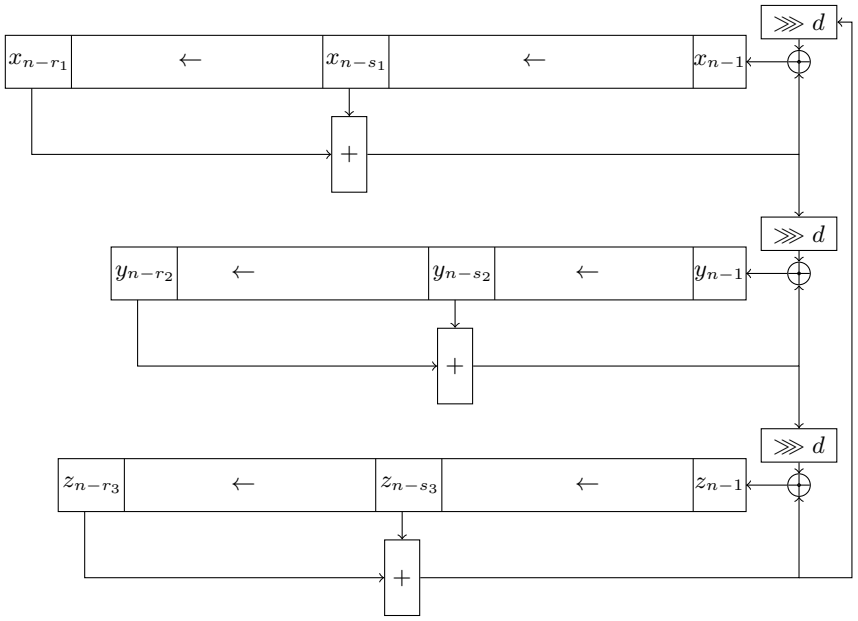


Fig. 2. Description of Trifork

The seed of the generator is $(X_{-r_1}, Y_{-r_2}, Z_{-r_3})$. To fill its internal states, it will use a Linear Congruential Generator of public parameters a, c, m with a odd and where m is the same as the one used by the LFGs.

For $i \in \{-r_1 + 1, \ldots, -1\}, X_i = aX_{i-1} + c \bmod m$
For $i \in \{-r_2 + 1, \ldots, -1\}, Y_i = aY_{i-1} + c \bmod m$
For $i \in \{-r_3 + 1, \ldots, -1\}, Z_i = aZ_{i-1} + c \bmod m$

A step i, the generator computes

$$\begin{aligned} X'_i &= X_{i-r_1} + X_{i-s_1} \bmod m \\ Y'_i &= Y_{i-r_2} + Y_{i-s_2} \bmod m \\ Z'_i &= Z_{i-r_3} + Z_{i-s_3} \bmod m \end{aligned}$$

$$X_i = X'_i \oplus (Z'_i \ggg d) \tag{27}$$
$$Y_i = Y'_i \oplus (X'_i \ggg d) \tag{28}$$
$$Z_i = Z'_i \oplus (Y'i \ggg d) \tag{29}$$

where d is a constant satisfying $0 < d < N$; $\oplus$ is the bitwise exclusive-or and $\ggg$ is the right-shift operator. The output at step n is:

$$W_n = X_n \oplus Z_n.$$

The security of Trifork is based on the secrecy of the internal states. We will present an algorithm that retrieves X_{-r_1}, Y_{-r_2} and Z_{-r_3} in 2^{64} steps.

5 Attack on Trifork

The reason this attack will use 2^{64} steps is because we start by guessing a third of the seed: X_{-r_1} of length 64 bits.

5.1 Recovering Z_{-r_3}

We consider a parameter f_1 that will be the number of outputs we will use to recover Z_{-r_3}. We will set this parameter later.

We denote by $\lceil X \rceil_d$ the d upper bits of a value, $\lfloor X \rfloor_d$ its d lower bits and consider the two following functions :

$$g : i \to \sum_{j=0}^{i-1} a^j \bmod m \text{ and } f : (r, s, i) \to g(r - s + i) + g(i) \bmod m$$

The first step is to compute an approximation of the d upper bits of the values $\{X_0, \ldots, X_{f_1-1}\}$. If $i < 0$, $X_i = a(...a(aX_{-r_1} + c) + c...) + c \bmod m$, that we conveniently rewrite $X_i = a^{r_1+i}X_{-r_1} + g(r_1 + i) \times c \bmod m$. If $i \geq 0$, by eq (27), $\lceil X_i \rceil_d = \lceil X_{i-s_1} + X_{i-r_1} \bmod m \rceil_d$.

- if $i < s_1$, then $\lceil X_i \rceil_d = \lceil a^i(1 + a^{r_1-s_1})X_{-r_1} + f(r_1, s_1, i) \times c \bmod m \rceil_d$ and we can compute this value correctly.
- if $i \geq s_1$, then $\lceil X_i \rceil_d \simeq \lceil X_i - r_1 \rceil_d + \lceil X_{i-s_1} \rceil_d = \lceil a^i X_{-r_1} + g(i) \times c \bmod m \rceil_d + \lceil X_{i-s_1} \rceil_d$ and we can only compute the $d - (i - s_1)$ upper bits correctly.

With that we obtain an approximation of the d upper bits of $\{Z_0, \ldots, Z_{f_1-1}\}$ knowing that $Z_i = W_i \oplus X_i$. We call these approximations $\bar{Z}_i$.

- if $i < s_3$, then $\lceil Z_i \rceil_d = \lceil a^i(1 + a^{r_3-s_3})Z_{-r_3} + f(r_3, s_3, i) \times c \bmod m \rceil_d$. We set $t_i = \bar{Z}_i 2^{n-d} - f(r_3, s_3, i) \times c$.
  - If $i < s_1$ then $\bar{Z}_i = \lceil Z_i \rceil_d$ and $a^i(1 + a^{r_3-s_3})Z_{-r_3} - t_i \bmod m = \lfloor Z_i \rfloor_{n-d}$. Hence $|a^i(1 + a^{r_3-s_3})Z_{-r_3} - t_i| < 2^{n-d}$.
  - If $i \geq s_1$, $\bar{Z}_i$ and $\lceil Z_i \rceil_d$ are only equal on the $d - (i - s_1)$ upper bits. Hence $|a^i(1 + a^{r_3-s_3})Z_{-r_3} - t_i| < 2^{n-d+i-s_1}$.
- if $i \geq s_3$, then $\lceil Z_i \rceil_d = \lceil a^i Z_{-r_3} + Z_{i-s_3} + g(i) \times c \bmod m \rceil_d$. We set $t_i = (\bar{Z}_i - \bar{Z_{i-s_3}})2^{n-d} - g(i) \times c$.
  - If $i < s_1$ then $\bar{Z}_i = \lceil Z_i \rceil_d$ and $\bar{Z_{i-s_3}} = \lceil Z_{i-s_3} \rceil_d$, so

$$\begin{aligned} a^i Z_{-r_3} - t_i = \; & a^i Z_{-r_3} - (\lceil Z_i \rceil_d - \lceil Z_{i-s_3} \rceil_d)2^{n-d} - g(i) \times c \bmod m \\ = \; & Z_{i-r_3} - (\lceil Z_i \rceil_d - \lceil Z_{i-s_3} \rceil_d)2^{n-d} \bmod m \\ = \; & (\lceil Z_{i-r_3} \rceil_d + \lceil Z_{i-s_3} \rceil_d - \lceil Z_{i-s_3} + Z_{i-r_3} \rceil_d)2^{n-d} \\ & + \lfloor Z_{i-r_3} \rfloor_{n-d} \bmod m` \end{aligned}$$

    Hence $|a^i Z_{-r_3} - t_i| < 2^{n-d+1}$.

- If $i \geq s_1$, $\bar{Z}_i$ and $\lceil Z_i \rceil_d$ are only equal on the $d-(i-s_1)$ upper bits. Hence $|a^i(1+a^{r_3-s_3})Z_{-r_3} - t_i| < 2^{n-d+i-s_1+1}$.

Remark 1 *As we use few outputs we will not treat the case were $i - r_3 > 0$.*

Case 1: If $s_3 \geq f_1$, we construct

$$T = (T_i)_{i<f_1},$$

which is close to

$$(1+a^{r_3-s_3})Z_{-r_3} \times (1, a, a^2, \ldots, a^{f_1-1}) \bmod m.$$

We can see T as the outputs of a *Truncated Linear Congruential Generator* of seed $(1+a^{r_3-s_3})Z_{-r_3}$ and known multiplier a. So we search for the closest vector to T in the lattice: $\{X \times (1, a, a^2, \ldots, a^{f_1-1}) \bmod m | X \in \mathbb{Z}\}$. This lattice is spanned by the line of the following matrix:

$$\begin{pmatrix} 1 & a & a^2 & \ldots & a^{f_1-1} \\ 0 & m & 0 & \ldots & 0 \\ 0 & 0 & m & \ldots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \ldots & m \end{pmatrix}.$$

The Closest Vector Problem (CVP) is finding, in a given lattice, the closest vector to a vector target T. This is usually a hard problem and we could have used the attack described in [8]. But here the matrix is of small dimension and we can solve exactly the CVP thanks to a CVP solver/ We use the CVP solver of the fpylll library [15] for python.
If f_1 is large enough the CVP solver returns

$$(1+a^{r_3-s_3})Z_{-r_3} \times (1, a, a^2, a^3, \ldots) \bmod m$$

. We obtain $(1+a^{r_3-s_3})Z_{-r_3}$ but not Z_{-r_3} because $(1+a^{r_3-s_3})$ is not invertible mod m.

Case 2: If $s_3 < f_1$, we set $b = a^{-1} \bmod m$ and $\alpha_3 = (1+a^{r_3-s_3})$. We construct

$$T = (t_{s_3}, \ldots, t_{f_1-1}, t_0, \ldots, t_{s_3-1})$$

which is close to

$$a^{s3}Z_{-r_3} \times (1, a, a^2, \ldots, a^{f_1-1-s_3}, b^{s_3}\alpha_3 \ldots, b\alpha_3) \bmod m.$$

We search for the closest vector to T in the lattice:

$$\{X \times (1, a, a^2, \ldots, a^{f_1-1-s_3}, b^{s_3}\alpha_3 \ldots, b\alpha_3) \bmod m | X \in \mathbb{Z}\}$$

. This lattice is spanned by the line of the following matrix:

$$\left(\begin{array}{cccc|cccc} 1 & a & \ldots & a^{f_1-1-s_3} & b^{s_3}\alpha_3 & b^{s_3-1}\alpha_3 & \ldots & b\alpha_3 \\ 0 & m & \ldots & 0 & 0 & 0 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & m & 0 & 0 & 0 & \ldots \\ 0 & 0 & \ldots & 0 & m & 0 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}\right).$$

If f_1 is large enough the CVP solver returns

$$a^{s_3} Z_{-r_3} \times (1, a, a^2, \ldots, a^{f_1-1-s_3}, b^{s_3}\alpha_3 \ldots, b\alpha_3) \bmod m$$

and we compute Z_{-r_3}.

The value f_1 is large enough when we have n bits of correct information. If $n/d < s_1$, then we set $f_1 = n/d + 1$ and the $d-1$ upper bits of the $n/d+1$ computed approximation of X_i are correct. If $n/d \geq s_1$ the we set f_1 such that $f_1 - 1 \times (d - f_1 - s_1) \geq n$.

If we set X_{-r_1}, we compute Z_{-r_3} or $\alpha_3 Z_{-r_3}$ by solving one CVP on a matrix of size $f_1 \times f_1$.

5.2 Recovering Y_{-r_2}

We consider a parameter f_3 that will be the number of outputs we will use to recover Y_{-r_2}. We will set this parameter as we set f_1. If $n/d < s_3$, then we set $f_3 = n/d+1$ and the $d-1$ upper bits of the n/d computed approximation of Z_i are correct. If $n/d \geq s_3$ the we set f_3 such that $f_3 - 1 \times (d - f_3 - s_3) \geq n$.

Firstly we will compute an approximation of the $n-d$ upper bits of the values $\{Z_0, \ldots, Z_{f_3-1}\}$.

- if $i < s_3$, then $\lceil Z_i \rceil_d = \lceil a^i(1 + a^{r_3-s_3})Z_{-r_3} + f(r_3, s_3, i) \times c \bmod m \rceil_d$ and we can compute this value correctly.
- if $i \geq s_3$, then $\lceil Z_i \rceil_d \simeq \lceil a^i Z_{-r_3} + g(i) \times c \bmod m \rceil_d + \lceil Z_{i-s_3} \rceil_d$ and only the $d - (i - s_3)$ upper bit are computed correctly.

Remark 2 *If we do not know Z_{-r_3} but only $(1 + a^{r_3-s_3})Z_{-r_3}$, it means $s_3 \geq f_1 \geq n/d$. So $f_3 = n/d$ and $s_3 \geq f_3$ and we never need Z_{-r_3}.*

Secondly we will compute an approximation of the $n - d$ lower bits of the values $\{X_0, \ldots, X_{f_3-1}\}$.

- if $i < s_1$, then $X_i = (a^i(1+a^{r_1-s_1})X_{-r_1} + f(r_1, s_1, i) \times c \bmod m) \oplus (Z_i \ggg d)$.
- if $i \geq s_1$, then $X_i = (a^i X_{-r_1} + g(i) \times c + X_{i-s_1} \bmod m) \oplus (Z_i \ggg d)$.

With the lower bits of the (X_i) we can compute an approximation of the $n - d$ lower bits of the values $\{Z_0, \ldots, Z_{f_3-1}\}$ knowing that $Z_i = W_i \oplus X_i$.

Then we obtain an approximation of the $n - d$ upper bits of $\{Y_0, \ldots, Y_{f_3-1}\}$ knowing that $Z_i = (Z_{i-r_3} + Z_{i-s_3} \bmod m) \oplus (Y_i \ggg d)$. We call these new values $\bar{Y_i}$.

Remark 3 *When we computed the upper bits of (Z_i), we only had the d upper bits, not the $n - d$. This lack of information impacts the rest of the calculation and at the final step, we know there is no information in the $n - 2d$ lower bits of the $(\bar{Y_i})$.*

- if $i < s_2$, then $\lceil Y_i \rceil_d = \lceil a^i(1+a^{r_2-s_2})Y_{-r_2} + f(r_2, s_2, i) \times c \bmod m \rceil_d$. We set $t_i = \bar{Y_i} 2^d - f(r_2, s_2, i) \times c$.

– if $i \geq s_2$, then $\lceil Y_i \rceil_d = \lceil a^i Y_{-r_2} + Y_{i-s_2} + g(i) \times c \bmod m \rceil_d$. We set $t_i = (\bar{Y_i} - \bar{Y_{i-s_2}})2^d - g(i) \times c$.

Here the dependences between the different values are harder to make explicit. For example, in the case where $i < min(s_1, s_2, s_3)$, we can compute the d upper bits of Z_i correctly. Thank to that we can compute the d upper bits of $\lfloor X_i \rfloor_{n-d}$ correctly.We obtain directly the d upper bits of $\lfloor Z_i \rfloor_{n-d}$ with $Z_i = W_i \oplus X_i$. The last step is obtaining the d upper bits of $Y_i \ggg d$. At this point there is an addition so we might loose one bit of precision because of a carry. We obtain that $|a^i(1 + a^{r_2-s_2})Y_{-r_2} - t_i| < 2^{n-d+1}$.

Case 1: If $s_2 \geq f_3$, we construct

$$T = (T_i)_{i<f_3}$$

which is close to $(1 + a^{r_2-s_2})Y_{-r_2} \times (1, a, a^2, \dots) \bmod m$. We search for the closest vector to T in the lattice:

$$\{X \times (1, a, a^2, \dots) \bmod m | X \in \mathbb{Z}\}$$

. This lattice is spanned by the lines of the following matrix:

$$\begin{pmatrix} 1 & a & a^2 & \dots \\ 0 & m & 0 & \dots \\ 0 & 0 & m & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}.$$

The CVP solver returns $(1 + a^{r_2-s_2})Y_{-r_2} \times (1, a, a^2, \dots) \bmod m$. We cannot compute Y_{-r_2} because $(1 + a^{r_2-s_2})$ is not invertible mod m.

Case 2: If $s_2 < f_3$, we set $b = a^{-1} \bmod m$ and $\alpha_2 = (1 + a^{r_2-s_2})$. We construct

$$T = (t_{s_2}, \dots, t_{f_3-1}, t_0, \dots, t_{s_2-1})$$

which is close to

$$a^{s2} Y_{-r_2} \times (1, a, a^2, \dots, a^{f_3-1-s_2}, b^{s_2}\alpha_2 \dots, b\alpha_2) \bmod m.$$

and we search for the closest vector to T in the lattice:

$$\{X \times (1, a, a^2, \dots, a^{f_3-1-s_2}, b^{s_2}\alpha_2 \dots, b\alpha_2) \bmod m | X \in \mathbb{Z}\}$$

. This lattice is spanned by the lines of the following matrix:

$$\left(\begin{array}{cccc|cccc} 1 & a & \dots & a^{f_3-1-s_2} & b^{s_2}\alpha_2 & b^{s_2-1}\alpha_2 & \dots & b\alpha_2 \\ 0 & m & \dots & 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & m & 0 & 0 & 0 & \dots \\ 0 & 0 & \dots & 0 & m & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}\right)$$

We CVP solver returns $a^{s_2} Y_{-r_2} \times (1, a, a^2, \dots, a^{f_3-1-s_2}, b^{s_2}\alpha_2 \dots, b\alpha_2) \bmod m \bmod m$ and we can compute Y_{-r_2}.

Once again, for a set X_{-r_1} we only solve one CVP to compute Y_{-r_2} or $\alpha_2 Y_{-r_2}$.

Remark 4 *We will not detail here how we recover Z_{-r_3} and/or Y_{-r_2} in the cases where we only have $(1 + a^{r_3 - s_3})Z_{-r_3}$ and/or $(1 + a^{r_2 - s_2})Y_{-r_2}$ because it does not make appears interesting techniques. It only use modular arithmetic and does not need other guess or resource-consuming operation.*

This attack is fully implemented in sagemath but cannot run on a laptop as it needs to solve $2^{64} \times 2$ CVPs.

References

1. NESSIE, the new european schemes for signatures, integrity and encryption. https://www.cosic.esat.kuleuven.be/nessie/ (2000)
2. eStream, the ECRYPT stream cipher project. https://www.ecrypt.eu.org/stream/project.html (2004)
3. Babbage, S., De Cannière, C., Lano, J., Preneel, B., Vandewalle, J.: Cryptanalysis of SOBER-t32. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 111–128. Springer, Heidelberg (Feb 2003). https://doi.org/10.1007/978-3-540-39887-5_10
4. Banegas, G.: Attacks in stream ciphers: A survey. Cryptology ePrint Archive, Report 2014/677 (2014), https://eprint.iacr.org/2014/677
5. Blanco, A.B., López, A.B.O., Muñoz, A.M., Martínez, V.G., Encinas, L.H., Martínez-Graullera, O., Vitini, F.M.: On-the-fly testing an implementation of arrow lightweight prng using a labview framework. In: International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2019) and 10th International Conference on EUropean Transnational Education (ICEUTE 2019). pp. 175–184. Springer (2019)
6. Bleichenbacher, D., Patel, S.: SOBER crytanalysis. In: Knudsen, L.R. (ed.) FSE'99. LNCS, vol. 1636, pp. 305–316. Springer, Heidelberg (Mar 1999). https://doi.org/10.1007/3-540-48519-8_22
7. Elaine Barker, J.K.: Recommendation for random number generation using deterministic random bit generators. Tech. Rep. NIST Special Publication (SP) 800-90A, Rev. 1, National Institute of Standards and Technology, Gaithersburg, MD (2015). https://doi.org/10.6028/NIST.SP.800-90Arl
8. Frieze, A.M., Kannan, R., Lagarias, J.C.: Linear congruential generators do not produce random sequences. In: 25th FOCS. pp. 480–484. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715950
9. Keery A. McKay, L.B.: Report on lightweight cryptography. Tech. Rep. NISTIR 8114, National Institute of Standards and Technology, Gaithersburg, MD (2017). https://doi.org/10.6028/NIST.IR.8114
10. Knuth, D.: Deciphering a linear congruential encryption. IEEE Transactions on Information Theory **31**(1), 49–52 (1985)
11. López, A.B.O., Encinas, L.H., Muñoz, A.M., Vitini, F.M.: A lightweight pseudorandom number generator for securing the internet of things. IEEE access **5**, 27800–27806 (2017)
12. Orue, A., Montoya, F., Hernández Encinas, L.: Trifork, a new pseudorandom number generator based on lagged fibonacci maps. Journal of Computer Science and Engineering **2**, 46–51 (2010)

13. Rose, G.G.: A stream cipher based on linear feedback over GF(2^8). In: Boyd, C., Dawson, E. (eds.) ACISP 98. LNCS, vol. 1438, pp. 135–146. Springer, Heidelberg (Jul 1998). https://doi.org/10.1007/BFb0053728
14. Stern, J.: Secret linear congruential generators are not cryptographically secure. In: 28th FOCS. pp. 421–426. IEEE Computer Society Press (Oct 1987). https://doi.org/10.1109/SFCS.1987.51
15. development team, T.F.: fpylll, a lattice reduction library for python (2016), https://github.com/fplll/fpylll, available at https://github.com/fplll/fpylll