Mathematical studies of arithmetical pseudo-random numbers generators

Florette Martinez

▶ **To cite this version:**

Florette Martinez. Mathematical studies of arithmetical pseudo-random numbers generators. Cryptography and Security [cs.CR]. Sorbonne Université, 2023. English. NNT : 2023SORUS222 . tel-04214869

HAL Id: tel-04214869
https://theses.hal.science/tel-04214869v1

Submitted on 22 Sep 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunication et Électronique (Paris)

Présentée par

Florette MARTINEZ

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Mathematical studies of arithmetical pseudo-random numbers generators

Thèse dirigée par **Damien Vergnaud**

soutenue publiquement le 04 Juillet 2023

après avis des **rapporteurs** :

Mme. Adeline ROUX-LANGLOIS	Chargée de Recherche, CNRS
M. Mehdi TIBOUCHI	Industriel, NTT (Japon)

devant le **jury** composé de :

M. Jean-Sébastien CORON	Professeur, Université du Luxembourg
Mme. María NAYA-PLASENCIA	Directrice de Recherche, INRIA Paris
Mme. Adeline ROUX-LANGLOIS	Chargée de Recherche, CNRS
M. Mehdi TIBOUCHI	Industriel, NTT (Japon)
M. Damien VERGNAUD	Professeur, Sorbonne Université
M. Vincent ZUCCA	Maître de conférences, Université de Perpignan

Remerciements

C'est le début de ce manuscrit et pourtant nous voilà dejà dans les crédits du générique de fin ! Je n'ai hélas pas la possibilité de faire en sorte que ces dizaines de feuilles de papier vous jouent une musique façon carte de voeux musicale. Alors chantez-vous une petite chanson et c'est parti !

Je tiens bien évidemment à commencer par remercier Damien Vergnaud, mon directeur de thèse. C'est quelqu'un d'une patience et d'une gentillesse incroyables qui m'a toujours encouragé. Il m'a présenté des problèmes pertinents et n'a pas hésité à me fournir les outils adéquats (et colorés) pour les résoudre. Je voudrais également remercier Charles Bouillaguet grâce à qui j'ai compris l'importance de l'implémentation des attaques cryptographiques et qui m'a présenté un des TP d'informatique les plus complets et immersifs que j'aie pu voir. Bien que j'aie eu une collaboration beaucoup plus restreinte avec lui, je tiens à remercier Jean-Sébastien Coron, qui m'a permis de comprendre un peu plus en profondeur comment fonctionnait l'attaque de Stern sur le générateur congruentiel linéaire. Je remercie aussi Vincent Zucca et María Naya-Plasencia qui ont pris du temps pour faire partie de mon comité de suivi de thèse. Merci au jury d'être ici aujourd'hui et aux rapporteurs d'avoir pris le temps de lire mon manuscrit en détail.

Je voudrais remercier également les personnes qui étaient là quand je suis arrivée même si nous avons rarement discuté de sciences: Jean-Claude Bajard, Anand, Jérôme et Thomas. Même si le covid et le départ des troisièmes années a vidé le bureau, toute une nouvelle bande a su lui donner une nouvelle vie, alors merci à Abdel, Thibauld, Samuel, Jules, Mickael, Orel et Ahmed. Merci aussi aux peluches, fléchettes et autres Nerfs qui ont su remplir les fins d'après-midi où la motivation n'était plus au rendez-vous.

Merci à Noé, William, Maxence, Adrien et Émeline pour les gartic phones du dimanche soir, parce que le covid c'était dur. Merci à ma mère chez qui j'ai passé mes confinements et à sa cuisine depuis laquelle j'ai présenté mon papier à CT-RSA. Merci à Bertille, ma soeur qui est passé par là avant moi et qui a su me donner des conseils avisés. Merci à Claire, Pierre-Alexandre et Solène, qui sont là depuis longtemps et dont le soutien m'est très précieux. Merci à Julia, que j'ai rencontrée au tout début de ma thèse et qui n'est jamais vraiment partie du bureau depuis.

Et puis merci à Ambroise pour les *petits pains* (on parle ici bien évidemment de chocolatines) et pour la relecture de ma thèse (entre autres).

Contents

I The Linear Congruential Generator **1**
0.1 Pseudo-Random Number Generators . . . 2
0.2 Cryptanalysis of Pseudo-Random Number Generators . . . 3
0.3 The Linear Congruential Generator . . . 3
0.4 The Lagged Fibonacci Generator and the Multiple Recursive Generator . . . 5
0.5 The Knapsack Generator . . . 5
0.6 Contributions of this Thesis . . . 6
0.6.1 Organization of this thesis . . . 8
0.7 Notations . . . 8

1 Euclidean lattices **9**
1.1 Norms for lattices . . . 10
1.2 About short vectors in a lattice . . . 12
1.2.1 Shortest Vector problem . . . 12
1.2.2 The Gaussian Heuristic . . . 12
1.2.3 The Closest Vector problem . . . 13
1.3 Lattice basis reduction algorithms . . . 15
1.4 Babai's rounding algorithm . . . 15
1.5 Coppersmith method . . . 16
1.5.1 A basic version of the method . . . 16
1.5.2 The complete version . . . 18

2 Attacks Against the Linear Congruential Generator **19**
2.1 Attacks when the multiplier and the modulus are known . . . 20
2.1.1 Recover the seed solving a Closest Vector Problem . . . 20
2.1.2 Attack from Frieze *et al.* . . . 21
2.2 Attacks when the multiplier is unknown . . . 23
2.2.1 Attack using the Coppersmith method . . . 23
2.2.2 Stern simplified Attack . . . 26
2.2.3 Knuth Attack on the Knuth generator . . . 31
2.3 Attacks when both the multiplier and the modulus are unknown . . . 31
2.3.1 Presentation of the attack . . . 31
2.3.2 Theoretical choice of the parameters . . . 32
2.4 Variants of the Linear Congruential Generator . . . 34
2.4.1 One output over two . . . 34

2.4.2 Upper bits truncated . . . 34

II Reducing Pseudo Random Number Generators to Linear Congruential Generators 36

3 The Permuted Congruential Generator 37
3.1 Presentation of the Generator . . . 38
3.2 Dealing with a noisy truncated Linear Congruential Generator . . . 39
3.2.1 Reconstruction in Low Dimension Using Babai's Rounding . . . 40
3.3 State Reconstruction for PCG64 With Known Increment . . . 41
3.4 State Reconstruction for PCG64 With Secret Increment . . . 44
3.4.1 Partial Difference Reconstruction . . . 45
3.4.2 Predicting all the Rotations . . . 46
3.4.3 Full Difference Reconstruction . . . 48
3.4.4 Complete State Reconstruction . . . 48
3.5 Implementation and Practical Results . . . 49
3.5.1 Known Increment . . . 50
3.5.2 Unknown Increment . . . 51

4 Attack on Trifork 53
4.1 Description of Trifork . . . 53
4.2 General idea behind the attack . . . 55
4.3 Recovering Z_{-r_3} . . . 56
4.4 Recovering Y_{-r_2} . . . 58
4.5 Experimental results . . . 60

5 Attack on the Fast Knapsack Generator 61
5.1 Description of the Fast Knapsack Generator . . . 61
5.2 Attacking an LCG with non consecutive pairs of outputs . . . 63
5.3 Attacking the Fast Knapsack Generator . . . 64
5.3.1 Attack with consecutive outputs(Coppersmith method) . . . 64
5.3.2 Attack with consecutive outputs (Stern method) . . . 66
5.3.3 Attack via Coppersmith method without consecutive outputs . . . 67

6 Multiple Recursive Generator 69
6.1 Recovering the seed solving a Closest Vector Problem . . . 69
6.1.1 Experimental results . . . 70
6.2 Retrieving the seed using the attack from Frieze *et al.* . . . 71
6.2.1 Experimental results . . . 72
6.3 Recovering the seed when the multiplier is unknown . . . 73
6.3.1 Link with the simplified Stern attack for the Linear Congruential Generator . 73
6.3.2 Theoretical parameters . . . 73
6.3.3 Experimental results . . . 74
6.4 Multiple Recursive Generator with secret modulus . . . 75
6.5 The particular case of Combined Multiple Recursive Generators (CMRG) . . . 75
6.5.1 Attack on the MRG32 . . . 76

6.5.2 The MRG32k3a by L'Écuyer . . . 78

III Attack on a combined generators **79**

7 A Generalization of the Knapsack Generator **80**
7.1 Generalized Subset-Sum Generator . . . 80
7.2 High-level description of the attack . . . 81
7.3 Preliminaries . . . 82
7.4 Finding "Good Triplets" . . . 84
7.4.1 A Simple Sub-Quadratic Algorithm to Find Good Triplets . . . 87
7.4.2 Sub-Quadratic Algorithm with Overwhelming Success Probability . . . 87
7.5 Practical Key-recovery Attack on von zur Gathen-Shparlinski Elliptic Knapsack Generator . . . 89
7.5.1 Attack on the Elliptic Subset Sum Generator . . . 89
7.5.2 Experimental Results . . . 91
7.6 Theoretical Key-recovery Attack on the Elliptic Knapsack Generator . . . 92
7.7 Practical Key-recovery Attack on the Subset Product Generator . . . 93
7.7.1 Description of the Attack . . . 93
7.7.2 Experimental Results . . . 94

8 Arrow **96**
8.1 About Lightweight Cryptographic . . . 96
8.2 Presentation of Arrow . . . 96
8.3 Attack on a first hardware version of Arrow . . . 98
8.4 Another hardware version of Arrow . . . 100
8.5 A software version of Arrow . . . 102

9 Conclusion and perspectives **105**

Part I

The Linear Congruential Generator

Introduction

0.1 Pseudo-Random Number Generators

Cryptography is a field of computer science and mathematics that deals with the study of techniques and algorithms for securing communication and data against unauthorized access, modification, or disclosure. Pseudo-random number generators play a critical role in modern cryptography as they allow for the creation of numbers that are unpredictable and can be used as the basis for cryptographic keys and other essential components of cryptographic algorithms. More precisely, a Pseudo-Random Number Generator (PRNG) is an (efficient) deterministic algorithm that generates a sequence of numbers that appear to be statistically random and unpredictable even if they are actually generated from a fixed (short) initial value, called the *seed*. The generated (pseudo-random) numbers can be used for numerous applications in cryptography (e.g. key generation, initialization vector generation for block cipher modes of operation, nonces generation in communication protocols such as SSL/TLS or SSH to prevent replay attacks of in signature schemes such as ECDSA, password salting, ...).

The *one-time pad* is a standard cryptographic technique for encrypting messages using a random key that is as long as the message itself. The key is generated using a truly random process and used once and never reused; to encrypt the message, the sender combines each plaintext character with the corresponding key character using modular addition. The one-time pad is unconditionally secure (meaning that an attacker who intercepts the ciphertext is unable to derive any information about the plaintext without the key.) but its practical implementation has some severe limitations. One can use a PRNG to generate a pseudo-random key from a short seed (and possibly some initialization vector) and combine it with the plaintext (or ciphertext) using modular addition. In many cryptographic attacks, the attacker has access to both the plaintext and the corresponding ciphertext generated by an encryption algorithm; in the context of an encryption scheme constructed using a PRNG to emulate a one-time pad, this means that the adversary is given access to the actual outputs of the generator and may try to deduce the seed used by the generator. This kind of attack is deemed a *key-recovery attack* and it can have devastating consequences (since they would allow an attacker to decrypt any encrypted messages that were encrypted using the compromised seed). Actually, to be considered secure in practice, a PRNG should achieve the *indistinguishability* security definition which is a measure of how difficult it is for an attacker to distinguish the output of the generator from a truly random sequence. In this thesis, we will analyze the security of several number-theoretic PRNG and we will present key-recovery attacks against them (showing that they cannot be considered secure and should not be used in practice).

An alternative suitable for some setting is to use “True” Random Number Generators that generates truly random numbers by measuring physical phenomena that are unpredictable and

random in nature (e.g. temperature of a CPU, movements of a computer mouse, ...). These generators can be more expensive and less efficient than deterministic software counterparts because they require specialized hardware to measure the physical phenomena used to generate random numbers. Moreover, they can be vulnerable to various types of attacks, such as physical attacks or environmental factors that can affect the measured physical phenomena. An interesting approach is to use a hybridisation between the two techniques and to continuously collect inputs from the physical source of randomness and to produce outputs that depend on the previous inputs using a PRNG. This class of algorithm is usually called a *pseudo-random number generator with input* [7, 23]. Such generators will not be studied in this thesis.

0.2 Cryptanalysis of Pseudo-Random Number Generators

Analysing the quality of randomness for a PRNG suited for cryptographic applications is natural as a failure in these PRNGs would lead to problematic security breaches. In 1997, Golic presented in [29] a first attack against A5/1, a standard stream cipher for GSM communication. The list of attacks against this standard and its successors can be found in [16]. In 2008, a bug in the OpenSSL package in Linux led to insufficient entropy gathering and to practical attacks on the SSH and SSL protocols [67].

Attacking a non-cryptographic PRNG is not irrelevant. Non-cryptographic PRNGs tend to be faster and lighter than their cryptographic counterparts. As they do not pretend to achieve some kind of security, they are less studied by cryptanalysts hence there might not exist any known attack against them. Because of that, one might be tempted to replace a strong but slow cryptographic PRNG with a faster non-cryptographic one. Breaking non-cryptographic PRNGs could deter anyone to use them outside of what they are made for. Such mistakes have already been made, for example for the website *Hacker news* and it leads to a real life attack, see[26] (this particular example will be discussed later in section 6.5).

Attacking a non-cryptographic PRNG is not only security-related. PRNGs can be used in numerical simulations and a hidden structure in a PRNG could cause bias in said simulation. In [25], Ferrenberg et al. ran classical Ferromagnetic Ising model Monte-Carlo simulations in specific cases where exact results were known, with different PRNGs. They observed that the choice of the PRNG had a significant impact on the outcome. For example, a given linear feedback shift register tent to give energy levels that were too low and a critical temperature that was too high.

0.3 The Linear Congruential Generator

The Lehmer Generator (the first known ancestor of the Linear Congruential Generator) was presented by Lehmer in 1949 in [41]. It is defined by the recurrence relation

$$x_{n+1} = ax_n \bmod N$$

where $\mathbf{x}$ is the sequence of pseudo-random values, a the multiplier and N a modulus of the form $2^n \pm 1$. At this point, there was no notion of *secure* pseudo-random number generator, as the internal state was directly output at each step. In this article there is no generic discussion of the period of such a generator.

In 1958, Thomson presented in [62] a variation of this generator, defined by the recurrence relation

$$x_{n+1} = (4k+1)x_n + k \bmod 2^\ell$$

where k is odd. The goal of this generator was still not to be secure as they were still outputting the whole internal states but its efficiency: the modulus was a power of two to accelerate computations on a binary machine and the period was proven to be 2^n.

Rotenberg presented in 1960 in [55] his version of the linear congruential generator defined by the recurrence

$$x_{n+1} = (2^a + 1)x_n + c \bmod 2^{35}$$

where c is odd. With the multiplier being of this particular form, the generator is even more efficient than the previous one. Once again the period is proven to be 2^{35} and numerical tests were run to compute the correlation between two consecutive outputs (and thus estimate the statistical quality of the produced randomness). The same year, in [21], Coveyou presented a way to theoretically compute the correlation.

A first mention of using a Linear Congruential Generator to encrypt data can be found in [36] in 1985. Of course only the leading bits of the sequence should be used or there would be no security at all. In this article, Knuth considers the variation of the LCG defined by the recurrence

$$x_{n+1} = ax_n + c \bmod 2^\ell$$

where the multiplier a satisfies $a \equiv 1 \bmod 4$ and the constant c satisfies $c \equiv 1 \bmod 2$. He proposed a first algorithm to recover the seed of such a generator when the multiplier and the constant are not known and the last ℓ bits are missing with time complexity $\mathcal{O}(\ell)$. This attack will be described in subsection 2.2.3.

The first use of lattice-base cryptographic techniques against Truncated LCG comes from Frieze, Hastad, Kannan, Lagarias, and Shamir in 1988 in [27]. They present a lattice-based attack to recover solutions of linear congruential systems. One of their application is the attack of a Truncated Linear Congruential Generator when the multiplier, constant, and modulus are known. The algorithm runs in polynomial time and returns the correct values for a very large window of parameters. The algorithm will be presented in subsection 2.1.2.

The most famous algorithm used to attack the Truncated LCG even when the multiplier, the constant, and the modulus are unknown was presented in the article *Secret linear congruential generators are not cryptographically secure*[59] by Stern in 1987. The title is self-explanatory, the window of parameters allowing the Truncated LCG to be used as a cryptographic PRNG is not wide enough. The algorithm presented in the paper will be described in the section 2.3.

Does the LCG have any advantage left? It remains one of the oldest and best-known PRNG. It is easy to understand and to implement and one can find online lists of optimal parameters. Because of that some popular programming languages have their version of the LCG as their usual ”rand” function: the Gnu C compiler or the Turbo Pascal compiler for Pascal. But it seems that the LCG does not produce good quality randomness for scientific applications such as Monte Carlo computation, see [46].

0.4 The Lagged Fibonacci Generator and the Multiple Recursive Generator

Many generalizations of the LCG were proposed to achieve better efficiency and unpredictability. In 1969, Knuth [37] presented an unpublished additive generator devised in 1958 by Mitchell and Moore and based on the recursive sequence defined by

$$x_n = x_{n-24} + x_{n-55} \bmod N, \text{ for } n \geq 55$$

where N is even, and where $x_0, ..., x_{54}$ are arbitrary integers not all even. This generator is very fast since it does not require any multiplication. It is inspired by the simplest recursive sequence in which x_n depends on more than one of the preceding values, namely the *Fibonacci sequence*, and the integers 24 and 55 used in the definition are commonly called *lags*. In the late 90s, Reeds and Mitchell developed an interesting variant of this algorithm for an early version of UNIX; it was used in Plan 9 and eventually as the basic random source in the Go programming language in 2008:

$$x_n = x_{n-273} + x_{n-607} \bmod (2^{63} - 1), \text{ for } n \geq 607,$$

where for the initial values $x_0, \ldots, x_{607}$, Go uses a vector generated using a LCG.

The *Lagged Fibonacci Generators* (LFG) have found numerous applications even if it has some strong statistical flaws and it outputs its full internal state, making it easy to predict after some (short) amount of time.

To increase the period and improve the statistical properties of the numbers output by an LCG, many works were devoted to proposing and analysing generators from higher-order linear recurrence, which are called Multiple Recursive Generators (MRGs). These generators are defined by a seed $(C_0, \ldots, C_{k-1})$ and the recursive relation

$$X_{n+k} = a_{k-1}X_{n+k-1} + \cdots + a_0X_n + c \bmod N$$

where $\mathbf{a} = (a_0, \ldots, a_{k-1})$ is the *multiplier*, c the *constant* and N the modulus. If the modulus N is a prime number, then the maximum period of the output sequences of such an MRG can be as large as $N^k - 1$. The LFG is a simple example of an MRG with zero values except for two values equal to 1 in the multiplier and a zero constant.

0.5 The Knapsack Generator

The *knapsack problem* is a NP-hard problem that was already studied in the 19th century. In this problem, we consider several objects with weights and values and we want to fill a knapsack for it to be as light and as valuable as possible. A variation of this problem is called the *Subset Sum Problem* where all the objects have the same value and we want to attain a precise weight for the whole knapsack. In other words we have n weights $\omega_0, \ldots, \omega_{n-1} \in \{0, \ldots, M\}$ and an integer s and we search for a binary vector $\mathbf{u} = (u_, \ldots, u_{n-1})$ such that

$$\sum_{i=0}^{n-1} u_i\omega_i = s.$$

In 1983 [39], Lagarias and Odlyzko presented a first algorithm to solve the Subset Sum Problem using lattice-based techniques as long as $M/2^n < 0.6463\ldots$, it will be quickly detailed in subsection 1.2.1 as an example of an instance of a Short Vector Problem. This result was improved by Coster, Joux, LaMacchia, Odlyzko, Schnorr and Stern in 1992 in [20] to obtain a correct algorithm for $M/2^n < 0.9408\ldots$. The problem is still considered hard if $M = 2^n$ and this is why Rueppel and Massey introduced the *Knapsack Generator* [56] in 1985 for cryptographic purposes using a modular Subset Sum. We consider n secret bits $u_0, \ldots, u_{n-1}$ and we extend them using a Linear Feedback Generator (a weak PRNG) to obtain a flow of pseudo-random bits. We also consider n secret weights $\omega_0, \ldots, \omega_{n-1} \in \{0, \ldots, 2^n\}$. At step i the Knapsack Generator computes

$$v_i = \sum_{j=0}^{n-1} u_{i+j}\omega_j \bmod 2^n$$

and output y_i which is the $n - \ell$ leading bits of v_i where ℓ is an independent parameter. In 2011, Knellwolf and Meier [35] presented the main attack against this generator. They used a guess-and-determine strategy coupled with lattice-based techniques to recover most of the key in relevant instances of the generator. In order to run said attack, they needed to guess all the n initial control bits. Hence their attack had a time complexity $\Omega(2^n)$. An equivalent algorithm to theirs will be presented as an example of a Closed Vector Problem in subsection 1.2.3. In 2009, von zur Gathen and Shparlinski presented the *Fast Knapsack Generator* that had a far smaller key and was sensibly faster but had not undergo a serious cryptanalysis. This generator will be studied in chapter 5. In [64], they also presented an elliptic version of this generator that will be studied in chapter 7 of this manuscript.

0.6 Contributions of this Thesis

Cryptanalysis of the Permuted Congruential Generators. The Permuted Congruential Generators are popular conventional (non-cryptographic) pseudo-random generators designed in 2014. They are used by default in the NumPy scientific computing package. Even though they are not of cryptographic strength, their designer stated that predicting their output should nevertheless be ”challenging”. We present a practical algorithm that recovers all the hidden parameters and reconstructs the successive internal states of the generator. This enables us to predict the next “random” numbers and output the seeds of the generator. We have successfully executed the reconstruction algorithm using 512 bytes of challenge input; in the worst case, the process takes 20 000 CPU hours. This reconstruction algorithm makes use of cryptanalytic techniques, both symmetric and lattice-based. In particular, the most computationally expensive part is a “guess-and-determine” procedure that solves about 252 instances of the Closest Vector Problem on a very small lattice.

These results were originally presented at the international conference FSE 2020 in *Practical seed-recovery for the PCG Pseudo-Random Number Generator* by Bouillaguet, Martinez, and Sauvage [15].

Cryptanalysis of Trifork. Trifork is a family of pseudo-random number generators described in 2010 by Orue, Montoya, and Hernández Encinas. It is based on three lagged Fibonacci generators and has been claimed as cryptographically secure. To prevent “guess-and-determine” attacks, Trifork uses very large internal states that are initialized using a linear congruential generator from a

secret seed made of three secret words of 64 bits. We present a lattice-based attack on Trifork and show that it cannot have more than 64 bits of security and that it is thus not cryptographically secure.

Cryptanalysis of Arrow. In 2017, López, Encinas, Muñoz, and Vitini presented a new family of lightweight pseudo-random number generators, which they called 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. The techniques used there are different from the ones used in the remainder of this thesis as they are not lattice-related.

These last two contributions were presented at the international conference ACNS 2022 in *Practical Seed-Recovery of Fast Cryptographic Pseudo-Random Number Generator* by Martinez [48].

Cryptanalysis of the Fast Knapsack Generator. The fast knapsack generator was introduced in 2009 by von zur Gathen and Shparlinski. It generates pseudo-random numbers very efficiently with strong mathematical guarantees on their statistical properties but its resistance to cryptanalysis was left open since 2009. We present lattice-based practical seed-recovery attacks against this generator that are surprisingly efficient when the proportion of truncated bits in relation to the internal states is not too large. Their complexities do not strongly increase with the size of parameters, only with the proportion of discarded bits.

Cryptanalysis of Combined Multiple Recursive Generators. A combined multiple recursive generators is a pseudo-random number generator based on combining two or more multiple recursive generators. L'Écuyer presented the general construction in 1996 and a popular instantiation called MRG32k3a in 1999. We present lattice-based practical seed-recovery attacks against this generator family. We use algebraic relations with the underlying algebraic generators to show that they are cryptographically insecure. We provide a theoretical analysis as well as efficient implementations.

These last two contributions were presented at the international conference CT-RSA 2022 in *Attacks on Pseudo Random Number Generators Hiding a Linear Structure* by Martinez [47].

Cryptanalysis of the Elliptic Knapsack Generator. In 2004, von zur Gathen and Shparlinski suggested a generalization of the knapsack pseudo-random generator in arbitrary abelian groups and proposed to use it with elliptic curves defined over (prime) finite fields. This generator provides strong mathematical guarantees on their statistical properties and the authors claimed that: "the only available attack on this generator is the brute force search over all parameters defining this generator". We first present an attack based on a search of combinatorial relations, a (limited) brute force search, and simple linear algebra to practically break the parameters proposed by von zur Gathen and Shparlinsk. We then extend this attack using the algebraic group law of the underlying Abelian group and latticed-based techniques for cases where this partial brute force search becomes prohibitive.

These results have not been published yet and are a joint work by Bouillaguet, Martinez, and Vergnaud.

0.6.1 Organization of this thesis

The first chapter presents an introduction to Euclidean lattices and their applications in cryptography. We will discuss their use in solving certain instances of the subset sum problem and in attacking the Knapsack Generator. Additionally, we will introduce several lattice-based tools such as the Gaussian Heuristic, Babai rounding algorithm, and Coppersmith method, which will be utilized in subsequent chapters.

Chapter 2 will focus on the Linear Congruential Generator and the known attacks against it, depending on the public parameters. In Chapter 3, we present our results on the Permuted Congruential Generator. Chapter 4 is dedicated to describing an attack against Trifork.

Chapters 5 and 6 analyze the Fast Knapsack Generator and the Multiple Recursive Generator, which can be viewed as a generalization of the LCG. We also present an attack against the Combinated Multiple Recursive Generators. Chapter 7 is devoted to our attacks on generalizations of the knapsack generator, including the elliptic Knapsack Generator and the last Chapter presents attacks against the lightweight generator Arrow.

0.7 Notations

Here are some useful notations that will be used in this whole manuscript

- Vectors, tuples or sequences will be denoted by bold letters like $\mathbf{v}$ and v_i denotes the i-th element of $\mathbf{v}$.
- We will denote by $\mathbf{v} \bmod N$ the vector $(v_0 \bmod M, v_1 \bmod M, \dots)$ and by $\mathbf{v}/a$ the vector $(v_0/a, v_1/a, \dots)$
- The XOR operation is denoted $\oplus$
- The integer division is denoted div
- Left and right rotations are denoted $\lll$ and $\ggg$ respectively.
- Left and right shift are denoted $\ll$ and $\gg$ respectively. Shifts are defined as in the programming language C, meaning that if x is an integer modulo 2^n then:

$$x \ll \ell = x \times 2^\ell \bmod 2^n \text{ and } x \gg \ell = x \operatorname{div} 2^\ell \bmod 2^n$$

- We will denote by $\mathcal{M}_{(n\times m)}(K)$ the set of matrices over the ring K with n rows and m columns. The ring K might be omitted if obvious.
- We will denote by $M_{i,j}$ the coefficient on the i-th row and j-th column of M.

Experimental results All the experimental results presented in this manuscript are averages of hundred instances of the algorithm run on sagemath v.9.5 on my laptop, a Dell Latitude running on Linux 22.04, unless otherwise specified. Theses codes are available on my git account.

`https://github.com/floretteM`

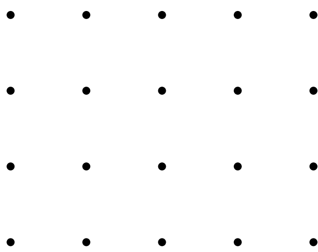
Chapter 1

Euclidean lattices

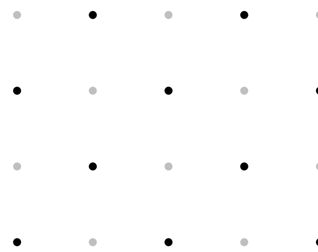
This chapter aims to introduce fundamental concepts, issues, and outcomes concerning lattices to facilitate the understanding of forthcoming chapters and help non-specialist readers.

Let $\mathbb{R}^n$ be the n-dimensional Euclidean space for some integer $n \geq 1$. An Euclidean lattice Λ of rank k and dimension d is a finite $\mathbb{Z}$-module of $\mathbb{R}^n$ of rank k. Figure 1.1 presents illustrations of three lattices of rank 2 in $\mathbb{R}^2$ and one lattice of rank 1.

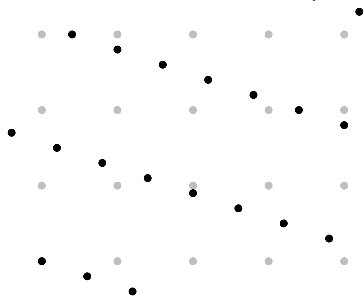
Λ_1:



Λ_2:



Λ_3:



Λ_4:

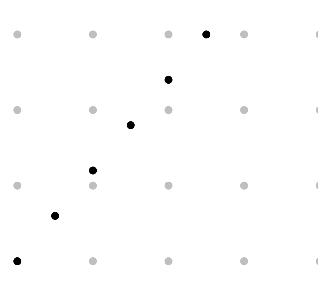


Figure 1.1: Four different lattices over $\mathbb{R}^2$(the grey points represent $\mathbb{Z}^2$).

A lattice Λ of rank k can be defined by a basis $\mathbf{B} = \{\mathbf{b}_0, \ldots, \mathbf{b}_{k-1}\}$ as

$$\Lambda = \left\{ \sum_{i=0}^{k-1} \alpha_i \mathbf{b}_i \text{ for } (\alpha_0, \ldots, \alpha_{k-1}) \in \mathbb{Z}^k \right\}.$$

For the lattices from Figure 1.1, one can see that:

- $\Lambda_1 = \{\alpha(0,1) + \beta(1,0) \text{ for } (\alpha, \beta) \in \mathbb{Z}^2\}$
- $\Lambda_2 = \{\alpha(0,2) + \beta(1,1) \text{ for } (\alpha, \beta) \in \mathbb{Z}^2\}$
- $\Lambda_3 = \{\alpha(0.2,1.5) + \beta(0.8,1.3) \text{ for } (\alpha, \beta) \in \mathbb{Z}^2\}$
- and $\Lambda_4 = \{\alpha(0.5,0.6) \text{ for } \alpha \in \mathbb{Z}\}$

Even if a lattice can be defined by a basis, this basis is not unique! We can define the lattice Λ_2 from the red basis or from the blue basis represented on Figure 1.2.

A lattice Λ can be represented by a matrix M whose lines are the vector $\mathbf{b}_i$'s. Hence Λ_2 can be represented by $\begin{pmatrix} 0 & 2 \\ 1 & 1 \end{pmatrix}$ or by $\begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix}$.

Proposition 1. *Let Λ be a lattice of rank n over $\mathbb{R}^n$ represented by a matrix $M \in \mathcal{M}_{(n\times n)}(\mathbb{R})$.*

- *For every matrix $Z \in \mathcal{M}_{(n\times n)}(\mathbb{Z})$ of determinant ± 1, the matrix $Z \times M$ also represent the lattice.*
- *Correspondingly, for every matrix $M' \in \mathcal{M}_{(n\times n)}(\mathbb{R})$ representing Λ there exists a matrix $Z \in \mathcal{M}_{(n\times n)}(\mathbb{Z})$ of determinant ± 1 such that $M' = Z \times M$.*

Corollary 1. *By the previous proposition, the absolute value of the determinant of a matrix M representing a lattice Λ is an invariant of the lattice called the* volume *and denoted* $\text{vol}(\Lambda)$*. In other words, if M represent Λ, then* $\text{vol}(\Lambda) = |\det(M)|$.

Definition 1. *If Λ is lattice and $\mathbf{B} = \{\mathbf{b}_0, \ldots, \mathbf{b}_{k-1}\}$ a basis, we call a fundamental domain the subset of space D defined by:*

$$D = \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{b}_i \mid \lambda_i \in [0,1[\right\}.$$

The volume of the fundamental domain is given by the determinant of the matrix associated to the basis $\mathbf{B}$.

As for basis, fundamental domains are not unique. In Figure 1.2 we draw two fundamental domains for the lattice Λ_2. One from the red basis and one from the blue basis.

1.1 Norms for lattices

In the following, the notions of "short" and "close" vectors are discussed. These notions are defined for a norm on $\mathbb{R}^n$.

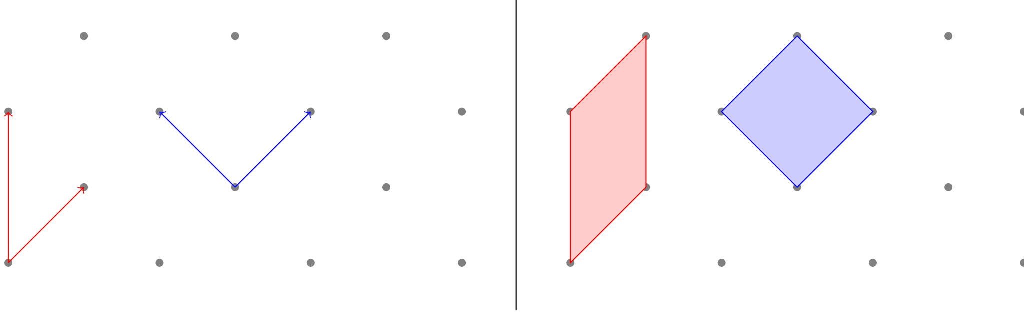


Figure 1.2: Two basis and their associated fundamental domains for the lattice Λ_2

Definition 2 (Euclidean norm). *Let $\mathbf{v} = (v_0, \ldots, v_{n-1})$ be a vector in $\mathbb{R}^n$, we define the euclidean norm of $\mathbf{v}$ as*

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=0}^{n-1} |v_i|^2}.$$

Definition 3 (Infinite norm). *Let $\mathbf{v} = (v_0, \ldots, v_{n-1})$ be a vector in $\mathbb{R}^n$, we define the euclidean norm of $\mathbf{v}$ as*

$$\|\mathbf{v}\|_\infty = \underset{i}{max} |v_i|.$$

We will also define a norm on the matrices.

Definition 4 (Operator norm). *We define the operator norm on $M \in \mathcal{M}_{(n\times n)}(\mathbb{R})$ as*

$$|||M||| = \underset{\mathbf{x}\in\mathbb{R}^n \setminus 0}{sup} \frac{\|M\mathbf{x}\|_2}{\|\mathbf{x}\|_2}.$$

By definition, for any $\mathbf{x} \in \mathbb{R}^n$,$\|M\mathbf{x}\|_2 \leqslant |||M||| \|\mathbf{x}\|_2$.

We could also define another operator norm from the infinite norm but it would be of no use in this manuscript.

Proposition 2.

- *The operator norm is a sub-multiplicative norm. If we consider $A \in \mathcal{M}_{(n\times n)}(\mathbb{R})$ and $B \in \mathcal{M}_{(n\times n)}(\mathbb{R})$ then*

$$|||AB||| \leq |||A||| \times |||B|||$$

- *The operator norm of $M \in \mathcal{M}_{(n\times n)}(\mathbb{R})$ is equal to the largest eigenvalue of M.*

Remark 1. *Here we defined the operator norm on the "matrix × vector" product. But the matrices M and M^T (its transpose) have the same eigenvalue hence the same operator norm. For any vector $\mathbf{x} \in \mathbb{R}^n$,$\|\mathbf{x}M\|_2 \leqslant \|\mathbf{x}\|_2 |||M|||$.*

Definition 5 (Condition number). *The condition number of an invertible matrix $M \in \mathcal{M}_{(n\times n)}$ is given by $|||M||| \times |||M^{-1}|||$. We denote it* cond$(M)$. *It is used to define how well a problem is conditioned (how much an error in the input will affect the output). As the operator norm is sub-multiplicative, the condition number of a matrix is always greater or equal to one.*

1.2 About short vectors in a lattice

1.2.1 Shortest Vector problem

Definition 6. *The* Shortest Vector Problem *(SVP) consists in finding, in a lattice, a non-zero vector* $\mathbf{v}$ *with the shortest norm. We denote by* $\lambda_1(\Lambda)$ *(or directly* λ_1*) the value* $\|\mathbf{v}\|_2$.

The SVP is a hard problem as all the known SVP-solver are exponential in time. But even if it is hard in general we will see several examples in this manuscript where finding the shortest vector in a specific, structured lattice is easy. We can reduce some mathematical problems used in cryptography to the SVP.

Example 1 (The Subset Sum)**.** *Let us consider the case of the subset sum presented in 1983 by Lagarias and Odlyzko in [39]: we have* n *public weights* $(\omega_0, \ldots, \omega_{n-1}) \in \{1, \ldots, M\}$*, a binary (or at least very small) secret* $\mathbf{u} = (u_0, \ldots, u_{n-1})$ *and an output* $y = \sum_{i=0}^{n-1} x_i \omega_i$*. Finding* $\mathbf{u}$ *is supposed to be hard. We consider a large integer* N *and the following matrix:*

$$A = \begin{pmatrix} 1 & 0 & & 0 & N\omega_0 \\ 0 & 1 & & 0 & N\omega_1 \\ & & \vdots & & \\ 0 & 0 & & 1 & N\omega_{n-1} \\ 0 & 0 & & 0 & Ny \end{pmatrix}$$

We notice that $\mathbf{z} = (u_0, \ldots, u_{n-1}, -1) \times A = (u_0, \ldots, u_{n-1}, 0)$ *is a short vector in the lattice spanned by the rows of* A*. If* N *is large enough, only the vectors ending by zero can be candidates as being the shortest vector. As we already choose* $\mathbf{u}$ *small, we can hope that* $\mathbf{z}$ *is the shortest vector of the lattice (it is the case when* $M/2^n < 0.6$ *as detailed in the same article). Finding the vector* $\mathbf{u}$ *is equivalent to solving the SVP in this particular lattice.*

The decisional SVP (knowing if a given vector is the shortest in a lattice) is *also* a hard problem. To know if a candidate vector in a lattice Λ has its chance to be the chosen one, we need to know an approximation of the value $\lambda_1(\Lambda)$.

1.2.2 The Gaussian Heuristic

Given the volume of a lattice, we can at least easily estimate the value λ_1 by the Gaussian Heuristic. Earlier we have see the notion of fundamental domain. If Λ is a lattice and D a fundamental domain of Λ, we can tile Λ with D, centred on each lattice point. We tile $\mathbb{R}^2$ with the blue fundamental domain of Λ_2 in Figure 1.3.

The *Gaussian heuristic* "predicts" that if Λ is a full-rank lattice and C is a "nice" measurable subset of $\mathbb{R}^n$, then the number of points of $\Lambda \cap C$ is roughly $\mathrm{vol}(C)/\mathrm{vol}(\Lambda)$. That is to say, if we can fit k fundamental domains in C, we can assume there are k points of the lattice in C.

For example we consider C a rectangle 2×3 in Λ_2. By the Gaussian heuristic we should have around six points of the lattice in C.

We see in Figure 1.4 that we do have six points in C if we count its border. But if we shift C, only four points would remain in C or its border.

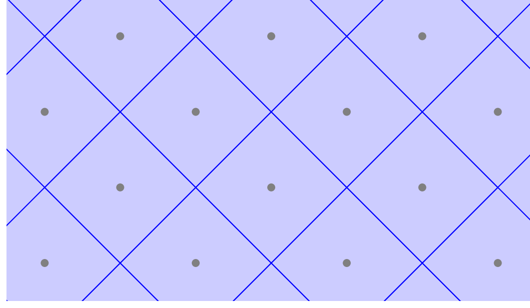


Figure 1.3: A fundamental domain of the lattice Λ_2 tiling $\mathbb{R}^2$

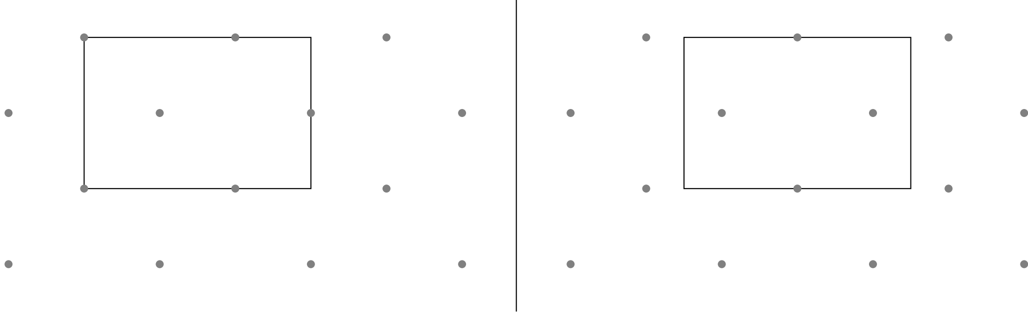


Figure 1.4: A subset C containing six points of lattice and its shift containing only 4 points.

Computing an approximation of λ_1: The Gaussian heuristic is neither precise nor proved but it is an intuition that will help us compute an approximation of λ_1. We fix C the $n-$ball of radius λ_1. Then $\Lambda \cap C$ should contain roughly 3 lattices points : 0, $\mathbf{v}$ a shortest vector and $-\mathbf{v}$. As the volume of the n-ball is $\frac{\pi^{n/2}}{\Gamma\left(\frac{n}{2}+1\right)}\lambda_1^n$, we obtain:

$$\lambda_1 = 3^{1/n} \times \mathrm{vol}(\Lambda)^{1/n} \left(\frac{\Gamma\left(\frac{n}{2}+1\right)}{\pi^{n/2}}\right)^{1/n}.$$

Using the Stirling formula, which is given by $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, we obtain

$$\lambda_1 \sim \frac{1}{\sqrt{2e\pi}}\sqrt{n}\,\mathrm{vol}(\Lambda)^{1/n}.$$

As it is a loose approximation and $\sqrt{2e\pi} \approx 4$ we will use the following:

$$\lambda_1 \approx \sqrt{n}\,\mathrm{vol}(\Lambda)^{1/n} \tag{1.1}$$

1.2.3 The Closest Vector problem

Definition 7. *The* Closest Vector Problem (CVP) *consists in finding, in a lattice, the closest vector to a certain target vector.*

In Figure 1.5, the red point is the closest lattice point to the blue target vector (the blue target vector does not have to be a lattice point).

Figure 1.5: Solving a CVP in the lattice Λ_2

The CVP is a hard problem and known CVP-solver are exponential in time. But we will see through this manuscript examples of particular, small or structured lattices where solving the CVP is easy. As for the SVP, there are mathematical problems used in cryptography that can be reduced to the CVP, such as retrieving the weight of the Knapsack Generator. We will present here an heuristic attack –different from the one of Knellwolf and Meier– against the Knapsack Generator. It seems to lead to similar results to the Knellwolf and Meier attack.

Example 2 (The Knapsack Generator)**.** *We consider n secret control bits $u_0, \dots, u_{n-1}$ that we extend in a secret pseudo-random flow $\mathbf{u}$ using a Linear Feedback Shift Register. Meaning we use a public binary polynomial P and computes u_{n+j} as $u_{n+j} = P(u_j, \dots, u_{j+n-1})$ for $j > 0$. We also consider n secret weights $\mathbf{w} = (\omega_0, \dots, \omega_{n-1}) \in \{0 \dots, 2^n\}$. At step j the Knapsack Generator computes*

$$v_j = \sum_{i=0}^{n-1} u_{i+j}\omega_i \bmod 2^n$$

and outputs y_j where $y_j = v_j \gg \ell$.

To attack, we start by guessing the n control bits $u_0, \dots, u_{n-1}$ so we know the pseudo-random flow $\mathbf{u}$. We denote by m the number of outputs and by $\mathbf{v}$ the vector $\mathbf{v} = (v_0, \dots, v_{m-1})$ which is in the lattice Λ spanned by the rows of the following matrix:

$$\begin{pmatrix} u_0 & u_1 & \dots & u_{m-1} \\ u_1 & u_2 & \dots & u_m \\ & & \iddots & \\ u_{n-1} & u_n & \dots & u_{n+m-2} \\ 2^n & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 2^n \end{pmatrix}$$

and close to $2^\ell \mathbf{y}$, where $\mathbf{y} = (y_0, \dots, y_{m-1})$.

We call $\mathbf{t}$ *the closest vector to* $2^\ell \mathbf{y}$ *in* Λ*. As the lattice contains very small vectors (of norm* $\simeq \sqrt{n}/2$*), there is no chance that* $\mathbf{v} = \mathbf{t}$*, but the two vectors will be really close. We denote by* $\boldsymbol{\omega}'$ *the vector satisfying* $\boldsymbol{\omega}'U = \mathbf{t}$*. To clearly define this vector we will need* U *to be of rank* n *so we must choose* m *a bit larger than* n*.*

If the control bits $u_0, \ldots, u_{n-1}$ *have been guessed correctly then we will have* $(\boldsymbol{\omega}' - \boldsymbol{\omega})U = (\mathbf{t} - \mathbf{v})$ *over* $\mathbb{Z}$ *and* $\|\boldsymbol{\omega}' - \boldsymbol{\omega}\|_2$ *small so we will be able to recover a good proportion of the secret weights. For* $n = 32$ *and* $m = 40$ *we present in the following table the percentage of bits recovered depending on the number of discarded bits (we consider that the secret control bits are already correctly guessed).*

ℓ	2	4	8	16	20	24	26	28
% of correct bits	97	92	80	56	42	30	20	3
computing time	$0.44s$	$0.43s$	$0.43s$	$0.45s$	$0.43s$	$0.43s$	$0.43s$	$0.44s$

1.3 Lattice basis reduction algorithms

The Lenstra–Lenstra–Lovász (LLL) algorithm is a polynomial time lattice-basis reduction algorithm invented by Lenstra, Lenstra, and Lovász in 1982, see [42]. Given a basis $\mathbf{B}$ of a lattice Λ, it computes a shorter basis $\mathbf{B}'$ of the same lattice in polynomial time.

This algorithm uses a parameter δ such that $0 < \delta < 1$ quantifying the quality of reduction. The closer to one δ is, the shorter the basis $\mathbf{B}'$ will be. The returned basis $\mathbf{B}' = \{\mathbf{b}'_0, \ldots, \mathbf{b}'_{m-1}\}$ satisfies several properties, in particular the following ones:

- The first vector in the basis cannot be much larger than the shortest non-zero vector: $\|\mathbf{b}'_0\|_2 \leq (2/(\sqrt{4\delta - 1}))^{n-1}\lambda_1$
- The first vector in the basis is also bounded by the determinant of the lattice: $\|\mathbf{b}'_0\|_2 \leq (2/(\sqrt{4\delta - 1}))^{(n-1)/2}(\det(\Lambda))^{1/n}$.

In practice the most common implementations of LLL use $\delta = 0.99$.

Remark 2. *If there is no vector* $\mathbf{v}$ *in the lattice satisfying* $\lambda_1 < \|\mathbf{v}\|_2 < (2/(\sqrt{4\delta - 1}))^{n-1}\lambda_1$*, then the LLL algorithm solves the exact SVP in polynomial time.*

1.4 Babai's rounding algorithm

In 1986, Babai proposed in [4] a simple algorithm that solves an approximate CVP. Let Λ be a full rank lattice represented by a LLL-reduced matrix $M \in \mathcal{M}_{(n \times n)}$. Then we can write:

$$\Lambda = \{\boldsymbol{\alpha} M | \boldsymbol{\alpha} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}^n\}.$$

Let $\mathbf{t}$ be our target vector. As M is invertible in $\mathbb{R}$, there exist $\boldsymbol{\beta} = (b_0, \ldots, b_{n-1}) \in \mathbb{R}^n$ such that $\boldsymbol{\beta} M = \mathbf{t}$. We denote by $\lceil \boldsymbol{\beta} \rfloor$ the vector $(\lceil b_0 \rfloor, \ldots, \lceil b_{n-1} \rfloor)$ where $\lfloor x \rceil$ denotes the nearest integer to x (using the "rounding half to even" tie-breaking rule). The vector $\lceil \boldsymbol{\beta} \rfloor$ is the closest integer vector to $\boldsymbol{\beta}$.

The vector $\lceil \boldsymbol{\beta} \rfloor \times M$ is a point of the lattice. Let $\mathbf{c}$ be the closest vector to $\mathbf{t}$ in Λ. Then

$$\begin{aligned}\|\lceil \boldsymbol{\beta} \rfloor \times M - \mathbf{c}\|_2 &\leq \|\lceil \boldsymbol{\beta} \rfloor \times M - \mathbf{t}\|_2 + \|\mathbf{t} - \mathbf{c}\|_2 \\ &\leq \|\lceil \boldsymbol{\beta} \rfloor \times M - \boldsymbol{\beta} \times M\|_2 + \|\mathbf{t} - \mathbf{c}\|_2 \\ &\leq \|\lceil \boldsymbol{\beta} \rfloor - \boldsymbol{\beta}\|_2 \times \|\!|M|\!\| + \|\mathbf{t} - \mathbf{c}\|_2\end{aligned}$$

A $\mathbf{c}$ is part of the lattice, $\mathbf{c}M^{-1}$ is an integer vector and its farther from $\boldsymbol{\beta}$ than $\lceil \boldsymbol{\beta} \rfloor$.

$$\begin{aligned}\|\lceil \boldsymbol{\beta} \rfloor \times M - \mathbf{c}\|_2 &\leq \|\mathbf{c}M^{-1} - \boldsymbol{\beta}\|_2 \times \|\!|M|\!\| + \|\mathbf{t} - \mathbf{c}\|_2 \\ &\leq \|\mathbf{c} - \mathbf{t}\|_2 \times \|\!|M^{-1}|\!\| \times \|\!|M|\!\| + \|\mathbf{t} - \mathbf{c}\|_2 \\ &\leq (\mathrm{cond}(M) + 1)\|\mathbf{c} - \mathbf{t}\|_2\end{aligned}$$

If $\mathbf{c}$ —the closest vector to $\mathbf{t}$ in Λ— satisfies $(\mathrm{cond}(M)+1)\|\mathbf{c}-\mathbf{t}\|_2 < \lambda_1$ then the Babai's rounding algorithm outputs $\mathbf{c}$.

Remark 3. *Why does M need to be LLL-reduced? Because an LLL-reduced matrix tends to have a shorter condition number than the original.[66]*

1.5 Coppersmith method

The Coppersmith method described here was first presented by Coppersmith in [19] and [18], we refer the reader to [32] for proofs. This algorithm, based on the LLL algorithm, aims to solve a multivariate modular polynomial system of equations.

1.5.1 A basic version of the method

We consider r linearly independent multivariate polynomials $P_1, \ldots, P_r$ defined over $\mathbb{Z}[z_0, \ldots, z_n]$, a secret small vector $\mathbf{x} = (x_0, \ldots, x_n)$ and a single known modulus N satisfying:

$$P_i(\mathbf{x}) \equiv 0 \bmod N \text{ for } i \in \{1, \ldots, r\}.$$

The vector $\mathbf{x}$ is said small in the sense that it must be bounded by known values, namely $|x_0| < X_0, \ldots, |x_n| < X_n$. To each of these polynomials P_i we associate a number k_i that will be the multiplicity of $\mathbf{x}$ as a root of P_i mod N (in other terms, k_i is the largest integer such that for all $k \leq k_i$, $P_i(\mathbf{x}) \equiv 0 \bmod N^k$). We construct the matrix $M \in \mathcal{M}_{(|\mathfrak{M}|+r\times|\mathfrak{M}|+r)}(\mathbb{R})$ as follows:

$$M = \left(\begin{array}{cccc|ccc} 1 & & & & & & \\ & X_0^{-1} & & & & \star & \\ & & \ddots & & & & \\ & & & X_0^{-a_0} \times \cdots \times X_n^{-a_n} & & & \\ \hline & & & & N^{k_1} & & \\ & & 0 & & & \ddots & \\ & & & & & & N^{k_r} \end{array}\right) \begin{array}{c} 1 \\ z_0 \\ \vdots \\ z_0^{a_0} \times \cdots \times z_n^{a_n} \\ \\ \\ \\ \end{array}$$

(columns $P_1 \downarrow \cdots P_r \downarrow$ above the right block)

We denote $\mathfrak{M}(= \{1, z_0, \ldots, z_n \ldots, z_0^{a_0} \times \cdots \times z_n^{a_n}\})$ the set of monomials that appear at least in one P_i and $|\mathfrak{M}|$ its cardinality. Each one of the upper rows (between 1 and $|\mathfrak{M}|$) corresponds to one of these monomials and each one of the latest columns (from $|\mathfrak{M}|+1$ to $|\mathfrak{M}|+r$) corresponds to one of the polynomials.

Let i be in $\{1, \ldots, |\mathfrak{M}|\}$, we denote m_i the i-th monomial of $\mathfrak{M}$, $m_i = z_0^{b_0} \ldots z_n^{b_n}$. The value of $M_{i,i}$ will be the inverse of the bound on m_i, hence $X_0^{-b_0} \ldots X_n^{-b_n}$. For all j between 1 and r, the value of $M_{i,|\mathfrak{M}|+j}$ will be the coefficient of m_i in P_j. Finally, the value of $M_{|\mathfrak{M}|+j,|\mathfrak{M}|+j}$ will be N^{k_j} as described in the previous paragraph.

Example 3. *We want to use this method to factor a RSA modulus $N = p \times q$ when the most significant bits of p and q are known. We call them p' and q' and we set*

$$P = (p' + z_0)(q' + z_1) = p'q' + q'z_0 + p'z_1 + z_0z_1.$$

This polynomial satisfies $P(x_0, x_1) \equiv 0 \bmod N$. The set of monomials is $\mathfrak{M} = \{1, z_0, z_1, z_0z_1\}$ and we construct the following matrix

$$M = \left(\begin{array}{cccc|c} 1 & & & & p'q' \\ & \frac{1}{X_0} & & & q' \\ & & \frac{1}{X_0} & & p' \\ & & & \frac{1}{X_0 \times X_0} & 1 \\ \hline & & 0 & & N \end{array}\right)$$

We want to show that the smallest vector of the lattice spanned by the rows of M contains the solution $\mathbf{x}$. We denote by c_i the integer such that $P_i(\mathbf{x}) = c_i N^{k_i}$. We can construct $\mathbf{v}$:

$$\begin{aligned} \mathbf{v} &= (1, x_0, \ldots, x_0^{a_0} \ldots x_{m-1}^{a_n}, -c_1, \ldots, -c_r) \times \mathcal{M} \\ &= \left(1, \frac{x_0}{X_0}, \ldots, \frac{x_0^{a_0} \ldots x_n^{a_n}}{X_0^{a_0} \ldots X_n^{a_n}}, 0, \ldots, 0\right). \end{aligned}$$

By construction, the vector $\mathbf{v}$ is in the lattice. Its first $|\mathfrak{M}|$ coordinates are smaller than one and the remaining ones are zero, hence it is a small vector. In general, retrieving the shortest vector of a lattice is a hard problem (called the SVP for Shortest Vector Problem), but if this short vector is abnormally short, it can be far easier. To obtain a small vector $\mathbf{v}$ we apply the LLL algorithm on M.

Remark 4. *This vector $\mathbf{v}$ might not be the smallest but the smallest satisfying $v_0 = 1$ and $(v_{m+1}, \ldots, v_{m+r}) = (0, \ldots, 0)$.*

The conditions on the bounds that make this method works are given by the following (simplified) equation:

$$\prod_{z_0^{b_0} \ldots z_n^{b_n} \in \mathfrak{M}} X_0^{b_0} \ldots X_n^{b_n} < N^{\sum_{i=1}^{r} k_i}. \tag{1.2}$$

For further details see [52].

1.5.2 The complete version

Instead of considering r linearly independent polynomials, we consider s irreducible and linearly independent polynomials $Q_1, \ldots, Q_s$ and we construct a new set of polynomials:

$$\mathcal{P} \subset \{z_0^{\alpha_0} \times \cdots \times z_{n-1}^{\alpha^{n-1}} Q_i | i \in \{1, \ldots, s\}, \boldsymbol{\alpha} \in \mathbb{Z}^n\}.$$

The polynomials in this new set still need to be linearly independent. This introduces an additional malleability as we can find a new and bigger family of polynomials but at the cost of a larger number of monomials.

Example. *We fix* $N = 27670116305985339737$, $p' = 2^{32}$, $q' = 2^{32} + 2^{31}$ *and* $X_0 = 32$. *If we consider the polynomial* $P = (p' + z_0)(q' + z_1) = p'q' + q'z_0 + p'z_1 + z_0 z_1$ *alone, the condition (1.2) is satisfied but it is a simplified version to understand the asymptotic behaviour of this method and the attack does not work. We now consider* $\mathcal{P} = \{P, z_0 P, z_1 P, P^2\}$. *The set of monomials becomes* $\mathfrak{M} = \{1, z_0, z_1, z_0 z_1, z_0^2, z_1^2, z_0^2 z_1, z_0 z_1^2, z_0^2 z_1^2\}$*and the equation (1.2) is still satisfied.*

$$
\begin{array}{c}
\begin{array}{cccc} P & z_0P & z_1P & P^2 \\ \downarrow & \downarrow & \downarrow & \downarrow \end{array} \\
M = \left(\begin{array}{ccccccccc|cccc}
1 & & & & & & & & & p'q' & 0 & 0 & (p'q')^2 \\
 & \frac{1}{X_0} & & & & & & & & q' & p'q' & 0 & 2p'(q')^2 \\
 & & \frac{1}{X_0} & & & & & & & p' & 0 & p'q' & 2(p')^2q' \\
 & & & \frac{1}{X_0^2} & & & & & & 1 & p' & q' & 4p'q' \\
 & & & & \frac{1}{X_0^2} & & & & & 0 & q' & 0 & (q')^2 \\
 & & & & & \frac{1}{X_0^2} & & & & 0 & 0 & p' & (p')^2 \\
 & & & & & & \frac{1}{X_0^3} & & & 0 & 1 & 0 & q' \\
 & & & & & & & \frac{1}{X_0^3} & & 0 & 0 & 1 & p' \\
 & & & & & & & & \frac{1}{X_0^4} & 0 & 0 & 0 & 1 \\
\hline
 & & & & \mathit{0} & & & & & N & & & \\
 & & & & & & & & & & N & & \\
 & & & & & & & & & & & N & \\
 & & & & & & & & & & & & N^2
\end{array}\right)
\begin{array}{c} 1 \\ z_0 \\ z_1 \\ z_0z_1 \\ z_0^2 \\ z_1^2 \\ z_0^2 \times z_1 \\ z_0 \times z_1^2 \\ z_0^2 \times z_1^2 \\ \\ \\ \\ \\ \end{array}
\end{array}
$$

The algorithm returns

$$\mathbf{v} = (1, \frac{15}{32}, \frac{23}{32}, \frac{345}{1024}, \frac{225}{1024}, \frac{529}{1024}, \frac{5175}{32768}, \frac{7935}{32768}, \frac{119025}{1048576}, 0, 0, 0, 0)$$

and we verify that $(2^{32} + 15)(2^{32} + 2^{31} + 23) = N$.

Chapter 2

Attacks Against the Linear Congruential Generator

The generator that we will present in this part is *the* Linear Congruential Generator (LCG), but there exists other PRNGs that are also congruential and linear and can be seen as a generalization of the LCG. A part of this chapter comes from the article *Attacks on Pseudo Random Number Generators Hiding a Linear Structure* presented at CT-RSA 2022 [47]. In particular the attack using a Coppersmith method is original (and a variation of it will be used in a following chapter).

We consider the LCG given by a seed x_0 and the equation

$$x_{i+1} \equiv ax_i + c \bmod N$$

where a is the multiplier, c the constant and N the modulus. To obtain the output y_i from an internal state x_i we truncate the ℓ lower bits. We denote the discarded bits by δ_i. We obtain $x_i = y_i 2^\ell + \delta_i$.

We first simplify the problem in two aspects.

Getting rid of the constant To work with a linear problem instead of an affine one, we want to make the constant c disappear. We consider the sequence (v_i) given by $v_i = x_{i+1} - x_i$. We know the most significant bits of each term of the sequence because $v_i = (y_{i+1} - y_i)2^\ell + (\delta_{i+1} - \delta_i)$ and $|(\delta_{i+1} - \delta_i)| < 2^{\ell+1}$. As this new sequence satisfies $v_{i+1} \equiv av_i \bmod N$, we have reduced the affine problem to a linear one. The reduction is not free as for now the $\ell + 1$ last bits are missing, instead of the ℓ last ones. From now we will only consider generators of the form $x_{i+1} \equiv ax_i \bmod N$.

Re centring the discarded bits The LCG is defined mod$N$. At each step i, the ℓ truncated bits form a value δ_i between 0 and $2^\ell - 1$. To recover this value, we will use lattice techniques. But these techniques operate on relative numbers. By centring the δ_i's around zero, we will lower the upper bound on them from 2^ℓ to $2^{\ell-1}$. The equation $x_i = y_i 2^\ell + \delta_i$ becomes

$$x_i = h_i + \delta'_i$$

where $h_i = y_i 2^\ell + 2^{\ell-1}$ and $\delta'_i \in \{-2^{\ell-1} + 1, \ldots, 2^{\ell-1} - 1\}$. From now we will only consider δ'_i and rename it δ_i.

Remark 5. *This two tips cannot be used simultaneously as getting rid of the constant will automatically recentre the discarded bits.*

Notation. *For a number m of outputs, we will use the following notations.*

$$\mathbf{x} = (x_0, x_1, \ldots, x_{m-1})$$

$$\mathbf{h} = (h_0, h_1, \ldots, h_{m-1})$$

$$\boldsymbol{\delta} = (\delta_0, \delta_1, \ldots, \delta_{m-1})$$

By definition, $\mathbf{x} = \mathbf{h} + \boldsymbol{\delta}$.

2.1 Attacks when the multiplier and the modulus are known

The internals states of the generator satisfy the following equation

$$x_{i+1} \equiv a x_i \bmod N$$

where a and N are public.

2.1.1 Recover the seed solving a Closest Vector Problem

Presentation of the attack

We consider the lattice Λ spanned by the lines of the following matrix.

$$L = \begin{pmatrix} 1 & a & a^2 & \ldots & a^{m-1} \\ & N & & & \\ & & N & & \\ & & & \ddots & \\ & & & & N \end{pmatrix}$$

The *unknown* vector $\mathbf{x}$ is part of this lattice as

$$\mathbf{x} \equiv x_0 \times (1, a, a^2, \ldots, a^{m-1}) \bmod N.$$

It is also close to the *known* vector $\mathbf{h}$ as $\mathbf{x} - \mathbf{h} = \boldsymbol{\delta}$ and $\|\boldsymbol{\delta}\|_\infty < 2^{\ell-1}$. We can expect to recover $\mathbf{x}$ by searching the closest vector to $\mathbf{h}$ in Λ using a CVP solver.

Theoretical choice of the parameters

We want $\mathbf{x}$ to be the closest vector to $\mathbf{h}$ in Λ. Let us suppose $\mathbf{x}'$ is the closest vector to $\mathbf{h}$ in Λ. Then

$$\begin{aligned} \|\mathbf{x} - \mathbf{x}'\|_2 &\leq \|\mathbf{x} - \mathbf{h}\|_2 + \|\mathbf{h} - \mathbf{x}'\|_2 \\ &\leq 2\|\mathbf{x} - \mathbf{h}\|_2 \\ &\leq 2\|\boldsymbol{\delta}\|_2 \\ &\leq 2^\ell \sqrt{m} \end{aligned}$$

The volume of the lattice Λ is N^{m-1}, thus by the Gaussian Heuristic we can assume that $\lambda_1 \simeq \sqrt{m}N^{(m-1)/m}$. If $\|\mathbf{x} - \mathbf{x}'\|_2 < \lambda_1$, we can expect $\mathbf{x}' = \mathbf{x}$ and $\mathbf{x}$ to be the closest vector to $\mathbf{h}$ thus returned by the CVP solver. The inequality can be simplified in

$$\ell < n \times (m-1)/m \tag{2.1}$$

where n denotes the size of N (that is to say $n \simeq \lceil log_2(N) \rceil$).

Complexity and limits

Algorithm 1 Seed retriever using a CVP solver

1: **procedure** AttackCVP($\mathbf{h}, a, N, m, \ell$)
2: $\quad L \leftarrow$ matrix($m \times m$) L described above.
3: $\quad \tilde{\mathbf{x}} \leftarrow$ CVP-solver($L, \mathbf{h}$)
4: $\quad$ **return** $\tilde{x}_0$ as a candidate for x_0.

This algorithm uses a CVP-solver on a matrix of size $m \times m$, its time complexity is exponential in m. With the formula (2.1), we see that every $l < n - 1$ should be reachable as $(m-1)/m$ tends to one.

Experimental results

For a given n and m we search for the greater ℓ such that the probability of success of retrieving the seed x_0 is above 50%.

m	2	3	4	5	16	32
$n = 32$						
ℓ (theoretical) $\leq$	16	21	24	25	30	31
ℓ (experimental) $\leq$	16	21	23	25	29	30
time	$0.0009s$	$0.001s$	$0.002s$	$0.003s$	$0.03s$	$0.11s$
$n = 64$						
ℓ (th.) $\leq$	32	42	48	51	60	62
ℓ (exp.) $\leq$	32	42	47	50	59	61
time	$0.001s$	$0.001s$	$0.002s$	$0.003s$	$0.03s$	$0.12s$
$n = 1024$						
ℓ (th.) $\leq$	512	682	768	819	960	992
ℓ (exp.) $\leq$	512	682	767	818	959	991
time	$0.001s$	$0.002s$	$0.003s$	$0.004s$	$0.06s$	$0.43s$

These results seem to confirm our heuristic. This algorithm is fast despite being exponential.

2.1.2 Attack from Frieze *et al.*

Presentation of the attack

Frieze *et al* described in [27] a method to solve a linear modular system when we already know a part of the solution. In the case of the LCG, the equations of the system are

$$a^i x_0 - x_i \equiv 0 \bmod N \text{ for } i \in \{1, \ldots, m-1\}.$$

A generator matrix for this system is:

$$A = \begin{pmatrix} N & & & & \\ a & -1 & & & \\ a^2 & & -1 & & \\ \vdots & & & \ddots & \\ a^{m-1} & & & & -1 \end{pmatrix}$$

The equation $A\mathbf{x} = 0 \bmod N$ has an infinite number of integer solutions. We apply the LLL-algorithm on A and obtain A' (it does not change the space of solution) and split $\mathbf{x}$ in $\mathbf{x} = \mathbf{h} + \boldsymbol{\delta}$. We denote by $\mathbf{c}$ the vector in $\{-N/2, \ldots, N/2\}^m$ satisfying $-A'\mathbf{h} \equiv \mathbf{c} \bmod N$. The new equation is $A'\boldsymbol{\delta} \equiv \mathbf{c} \bmod N$.

If $\|A'\boldsymbol{\delta}\|_\infty < N/2$, the equation is not modular any more and we can compute $\boldsymbol{\delta}$ as $(A')^{-1}\mathbf{c}$.

Theoretical choice of the parameters

Proved parameters The article of Frieze *et al.* contains the following theorem (Theorem 3.1)

Theorem 1. *For square-free $N > c(\epsilon, m)$ there is an exceptional set $E(N, \epsilon, m)$ of multipliers of cardinality $|E(N, \epsilon, m)| \leq N^{1-\epsilon}$ such that for any multiplier not in $E(N, \epsilon, m)$ the following is true. The x_i are uniquely determined by the knowledge of the $(1/m + \epsilon)\log_2(N) + m/2 + (m-1)\log_2(3) + 7/2\log(m) + 2$ leading bits of the x_i.*

Furthermore, there is an algorithm which runs in polynomial time in $\log_2(M) + m$ and finds $\mathbf{x}$

Heuristic parameters

- If A' represents one of the shortest basis of the lattice, by the Gaussian heuristic, its coefficients are roughly $N^{1/m}$. The inequality $\|A'\boldsymbol{\delta}\|_\infty < N/2$ becomes $\ell \leq n\frac{m-1}{m} - \log_2(m)$.
- If A' is the LLL-transformation of A, we know its coefficients will be slightly bigger.

Complexity and limits

This algorithm calls the LLL algorithm ($O(m^5n^3)$) and a matrix solver ($O(m^3)$) hence the time complexity of this algorithm is polynomial in m.

Algorithm 2 Seed retriever using lattice basis-reduction

1: **procedure** AttackFrieze($\mathbf{h}, a, N, m, \ell$)
2: $A' \leftarrow$ LLL-reduction of matrix A described above.
3: $\mathbf{k} \leftarrow \lfloor(-A' \times \mathbf{h})/N\rceil$
4: $\mathbf{c} \leftarrow -A'\mathbf{h} - N \cdot \mathbf{k}$
5: $\tilde{\boldsymbol{\delta}} \leftarrow (A')^{-1}\mathbf{c}$
6: $\tilde{\mathbf{x}} \leftarrow \mathbf{h} + \tilde{\boldsymbol{\delta}}$
7: **return** $\tilde{x}_0$ as a candidate for x_0.

Experimental results

For a given n and m we search for the greater ℓ such that the probability of success of retrieving the seed x_0 is above 50%.

m	2	3	4	5	16	32
$n = 32$						
ℓ (proved) $\leq$	7	9	8	6	0	0
ℓ (heuristic) $\leq$	17	21	24	25	26	26
ℓ (experimental) $\leq$	16	21	23	25	28	29
time	$0.001s$	$0.002s$	$0.003s$	$0.004s$	$0.04s$	$0.15s$
$n = 64$						
ℓ (p.) $\leq$	23	30	32	32	12	0
ℓ (h.) $\leq$	33	43	48	50	56	57
ℓ (exp.) $\leq$	32	42	47	50	58	60
time	$0.001s$	$0.002s$	$0.003s$	$0.004s$	$0.04s$	$0.15s$
$n = 1024$						
ℓ (p.) $\leq$	503	670	752	800	912	907
ℓ (h.) $\leq$	513	683	768	818	956	987
ℓ (exp.) $\leq$	512	682	767	818	958	990
time	$0.002s$	$0.002s$	$0.006s$	$0.006s$	$0.07s$	$0.32s$

Once again the results seem to confirm our heuristic (they are even a tad greater). As expected the attack is slightly less efficient (we attained smaller ℓ), but faster. In the case ($n = 1024$, $m = 32$) we go from 0.46s in the CVP attack to 0.32s in the Frieze attack.

2.2 Attacks when the multiplier is unknown

The internal states of the generator satisfy the following equation

$$x_{i+1} \equiv a x_i \bmod N$$

where the multiplier a is secret and the modulus N is public.

2.2.1 Attack using the Coppersmith method

Presentation of the attack

Let x_0, x_1, x_2 be 3 consecutive internal states of the LCG. We have $x_1 \equiv a x_0 \bmod N$ and $x_2 \equiv a x_1 \bmod N$. If a and N are coprime, we obtain:

$$x_1^2 \equiv x_0 x_2 \bmod N.$$

We replace x_i by $h_i + \delta_i$:

$$h_1^2 + 2h_1\delta_1 + \delta_1^2 = h_0h_2 + h_0\delta_2 + h_2\delta_0 + \delta_0\delta_2 \bmod N,$$

and notice that $(\delta_0, \delta_1, \delta_2)$ is a small root of the polynomial $P \bmod N$ where

$$P(z_0, z_1, z_2) = z_1^2 - z_0z_2 + 2h_1z_1 - h_0z_2 - h_2z_0 + h_1^2 - h_0h_2.$$

We can generalize this method. Let $x_0, \dots, x_k$ be $k+1$ consecutive internal states. We will obtain $\binom{k}{2}$ equations of the form $x_j x_{i+1} \equiv x_i x_{j+1} \bmod N$. Hence we will construct $\binom{k}{2}$ polynomials P_i of which $(\delta_0, \dots, \delta_k)$ is a simple root mod N.

Theoretical choice of the parameters

In the case with three outputs, we apply the Coppersmith method on P with bounds $X_0 = X_1 = X_2 = 2^\ell$. The set of monomials is $\mathfrak{M} = \{z_0, z_1, z_2, z_1^2, z_0 z_2\}$hence we should heuristically recover the root if $X_0 \times X_1 \times X_2 \times X_1^2 \times X_0 X_2 < N$, that is to say if $\ell/n < 1/7$.

In the generalization, the set of appearing monomials will be:

$$\mathfrak{M} = \{z_i | i \in \{0, \dots, k\}\} \bigcup \{z_i z_{j+1} | i, j \in \{0, \dots, k-1\}, i \neq j\}.$$

We find that $\prod_{z_i | i \in \{0,\dots,k\}} X_i \times \prod_{z_i z_{j+1} | i,j \in \{0,\dots,k-1\}, i \neq j} X_i X_{j+1} = (2^\ell)^{\Gamma(k)}$ where $\Gamma(k) = (k+1) + 2 \times 2\binom{k}{2}$. Thus, by eq.(1.2), the attack should work as long as $\ell/n < \binom{k}{2}/\Gamma(k)$. This theoretical bound increases toward 1/4.

An improvement of the Coppersmith method ?

We saw in section 1.5 that we could artificially increase the number of polynomials, which may lead to more favourable parameters.

For the reader familiar with [9] by Benhamouda et al., we will use the same notations. We denote $\mathcal{P}$ the bigger set constructed from the P_i. The polynomials in $\mathcal{P}$ are of the form $f = y_0^{k_0}, \dots, y_n^{k_n} P^{k_p}$ and are all linearly independent. We denote by $\chi_{\mathcal{P}}(f)$ the multiplicity of our small root as a root of f mod N: $\chi_{\mathcal{P}}(f) = k_p$. We denote $\mathfrak{M}$ the set of all the monomials appearing in $\mathcal{P}$. If m in $\mathfrak{M}$ is of the form $y_0^{k_0} \dots y_n^{k_n}$, we denote $\chi_{\mathfrak{M}}(m) = k_0 + \dots + k_n$. We know by equation (1.2) that the attack is supposed to work as long as

$$\ell/n \leq \frac{\sum_{f \in \mathcal{P}} \chi_{\mathcal{P}}(f)}{\sum_{m \in \mathfrak{M}} \chi_{\mathfrak{M}}(m)}$$

where ℓ is the number of discarded bits and n the size of the internal states of our generator.

Here our polynomial is $P = y_1^2 + 2H_1 y_1 + H_1^2 - y_0 y_2 - H_0 y_2 - H_2 y_0 - H_0 H_2$. We fix a parameter T and choose $\mathcal{P}_T$ as following:

$$\mathcal{P}_T = \{y_0^{k_0} y_1^{\epsilon} y_2^{k_2} P^{k_p} | \epsilon \in \{0,1\}, k_0 + \epsilon + k_2 + 2k_p \leq T\}$$

All the polynomials in $\mathcal{P}_T$ are linearly independent. Indeed, if we consider the monomial order $y_1 > y_0 > y_2$ then the leading monomial of $y_0^{k_0} y_1^{\epsilon} y_2^{k_2} P^{k_p}$ is $y_1^{2k_p+\epsilon} y_0^{k_0} y_2^{k_2}$ thus all leading monomials are different.

We are not going to precisely compute the set of monomial of $\mathcal{P}_T$ instead we are going to approach it with

$$\mathfrak{M}_T = \{y_0^{k_0} y_1^{k_1} y_2^{k_2} | k_0 + k_1 + k_2 \leq T\}.$$

Now we must compute $\sum_{f \in \mathcal{P}_T} \chi_{\mathcal{P}_T}(f)$ and $\sum_{m \in \mathfrak{M}_T} \chi_{\mathfrak{M}_T}(m)$:

$$\sum_{f\in\mathcal{P}_T}\chi_{\mathcal{P}_T}(f) = \sum_{k_0=0}^{T-2}\sum_{\epsilon=0}^{1}\sum_{k_2=0}^{T-2-k_0-\epsilon}\sum_{k_p=1}^{\lfloor\frac{T-k_0-\epsilon-k_2}{2}\rfloor} k_p = \lfloor\frac{((T+1)^2-1)\times((T+1)^2-3)}{48}\rfloor$$

$$\sum_{m\in\mathfrak{M}_T}\chi_{\mathfrak{M}_T}(m) = \sum_{k_0=0}^{T}\sum_{k_1=0}^{T-k_0}\sum_{k_2=0}^{T-k_0-k_1} k_0+k_1+k_2 = \frac{T(T+1)(T+2)(T+3)}{8}.$$

Thus this new construction should allow us to recover the small root as long as

$$\ell/n \leq \lfloor\frac{((T+1)^2-1)\times((T+1)^2-3)}{48}\rfloor \times \frac{8}{T(T+1)(T+2)(T+3)}.$$

This value tends to 1/6.

To obtain a bound bigger than 1/7 (our already achieved result), we need $T \geq 13$. But $T = 13$ means our lattice would be of dimension 924, and running the LLL algorithm on a lattice of dimension 900 is hardly doable.

Complexity and limits

Algorithm 3 Seed retriever using Coppersmith method

```
1:  procedure AttackCoppersmith(h, N, m, ℓ)
2:      lMono ← [1, z_0, ..., z_{m−1}]                      ▷ z_0, ..., z_{m−1} are variables
3:      lPoly ← []                                           ▷ We initialize an empty list
4:      for i in {1, ..., m − 1} do
5:          for j in {i + 1, ..., m − 1} do
6:              P ← z_i z_{j−1} − z_{i−1} z_j + h_i z_{j−1} + h_{j−1} z_i − h_j z_{i−1} − h_{i−1} z_j + (h_i h_{j−1} − h_{i−1} h_j)
7:              lPoly ← lPoly + [P]
8:              lMono ← lMono + [z_i z_{j−1}, z_{i−1} z_j]
9:      v ← Coppersmith(list_poly, list_mono, N, ell − 1, 1)
10:     if v vector in ∈ ℤ then
11:         x_0 ← h_0 + v_1
12:         return x_0
```

This algorithm calls the LLL algorithm on a matrix of size $2\cdot\binom{m-1}{2}+m$ hence the time complexity of this algorithm is exponential in m. We notice it does not recover the multiplier a but could easily do with few modifications.

This theoretical bound $\ell/n < \binom{k}{2}/\Gamma(k)$ increases toward 1/4. We cannot prove this attack can recover seeds if more than a quarter of the bits are discarded.

Experimental results

For a given n and m we search for the greater ℓ such that the probability of success of retrieving the seed x_0 is above 50%.

m $(= k+1)$	3	4	5	6	7
matrix size	5	10	17	26	37
$n = 32$					
ℓ (th.) $\leq$	4	6	6	6	7
ℓ (exp.) $\leq$	6	9	11	12	13
time	$0.008s$	$0.03s$	$0.08s$	$0.19s$	$0.39s$
$n = 64$					
ℓ (th.) $\leq$	9	12	13	13	14
ℓ (exp.) $\leq$	32	19	22	24	25
time	$0.008s$	$0.03s$	$0.08s$	$0.19s$	$0.41s$
$n = 1024$					
ℓ (th.) $\leq$	146	192	211	222	229
ℓ (exp.) $\leq$	204	307	361	393	415
time	$0.01s$	$0.04s$	$0.16s$	$0.54s$	$1.7s$

The heuristic gives only a lower bound on the attainable ℓ. This was expected as Coppersmith methods tend to be more effective in practice that in theory.

2.2.2 Stern simplified Attack

In this subsection we will present an alternate version of the attack presented by Stern in [59]. In this simplified version the modulus N is known. We denote by n the size of N $n = \lceil \log_2(N) \rceil$).

Presentation of the attack

We consider a new integer parameter d and a matrix seen earlier

$$M_1 = \begin{pmatrix} N & & & & \\ a & -1 & & & \\ a^2 & & -1 & & \\ \vdots & & & \ddots & \\ a^{d-1} & & & & -1 \end{pmatrix}$$

This matrix is unknown as the multiplier a is secret. The lattice Λ_1 spanned by the lines of M_1 contains vectors $(\mu_0, \ldots, \mu_{d-1})$ such that

$$\sum_{i=0}^{d-1} \mu_i a^i \equiv 0 \bmod N. \quad (2.2)$$

Let $\boldsymbol{\mu} = (\mu_0, \ldots, \mu_{d-1})$ be such a small vector. We obtain

$$\sum_{i=0}^{d-1} \mu_i h_{i+j} = \sum_{i=0}^{d-1} \mu_i x_{i+j} - \sum_{i=0}^{d-1} \mu_i \delta_{i+j}$$
$$\equiv \sum_{i=0}^{d-1} \mu_i a^i x_j - \sum_{i=0}^{d-1} \mu_i \delta_{i+j} \bmod N$$

$$\sum_{i=0}^{d-1} \mu_i h_{i+j} \equiv - \sum_{i=0}^{d-1} \mu_i \delta_{i+j} \bmod N \tag{2.3}$$

As $\boldsymbol{\mu}$ and the $\boldsymbol{\delta}$ are small, we can expect to have $|\sum_{i=0}^{d-1} \mu_i \delta_{i+j}|$ smaller than N.

We consider now a second integer parameter r and the the lattice Λ_2 spanned by the lines of the following matrix

$$M_2 = \left(\begin{array}{cccc|ccc} 2^{\ell-1} & & & & & \mathbf{h}_0 & \\ & 2^{\ell-1} & & & & \mathbf{h}_1 & \\ & & \ddots & & & \vdots & \\ & & & 2^{\ell-1} & & \mathbf{h}_{d-1} & \\ \hline & & & & N & & \\ & & 0 & & & \ddots & \\ & & & & & & N \end{array} \right)$$

where $\mathbf{h}_i = (h_i, h_{i+1}, \ldots, h_{i+r-1})$.

Let $\boldsymbol{\mu}$ be a short vector in Λ_1. By eq.(2.3), we know that there exists a vector $\mathbf{v}$ is in Λ_2 such that:

$$\mathbf{v} = (\mu_0 2^{\ell-1}, \ldots, \mu_{d-1} 2^{\ell-1}, -\sum_{i=0}^{d-1} \mu_i \delta_i, \ldots, -\sum_{i=0}^{d-1} \mu_i \delta_{i+r-1}).$$

As $\boldsymbol{\mu}$ and the δ_is are small, $\mathbf{v}$ will be a short vector in Λ_2. We retrieve $\mathbf{v}$ and thus $\boldsymbol{\mu}$ applying the LLL algorithm on M_2. By eq.(2.2), we obtain a polynomial P in one variable of degree $d-1$ such that $P(a) \equiv 0 \bmod N$.

If we redo all this algorithm again, we obtain a second polynomial Q such that $Q(a) \equiv 0 \bmod N$. The GCD of P and Q mod N, if of degree 1, should give the root a. Because of the use of a GCD, this attack mainly work with N **prime**.

Theoretical choice of the parameters

For a given r and d, the number of outputs needed to obtain one polynomial is $m_p = r + d - 1$. The matrix M_1 is of determinant N and of dimension d, if $\boldsymbol{\mu}$ is a short vector in Λ_1, we can expect $\|\boldsymbol{\mu}\|_\infty \simeq N^{1/d}$ by the Gaussian Heuristic. Still using the Gaussian Heuristic, we know that the norm of an average short vector in Λ_2 is $\sqrt{r+d}(2^{(\ell-1)d} N^r)^{\frac{1}{r+d}}$. The norm of $\mathbf{v}$ is close to $\sqrt{d + rd^2} 2^{\ell-1} 2^{n/d}$, thus we need the following inequality:

$$\sqrt{d + rd^2} 2^{\ell-1} 2^{n/d} < \sqrt{r+d}(2^{(\ell-1)d} 2^{nr})^{\frac{1}{r+d}}.$$

Experimentally, it seems that $r = d$ is quite optimal. If $m_p + 1$ is even we fix $r = d = (m_p + 1)/2$ and we obtain $\ell < 2 + n(1 - 2/d) - \log_2(d^2 + 1)$. If m_p is even we fix $d = m/2$ and $r = d + 1$ and we obtain $\ell < 1 + \frac{2d+1}{d+1}\left(\frac{1}{2}\log_2(\frac{2d+1}{d+d^2+d^3}) + n\frac{d+1}{2d+1} - \frac{n}{d}\right)$

Complexity and limits

Algorithm 4 Seed retriever using Stern method while N is known

```
 1: procedure FindPolynomialSimple(h, N, ℓ)
 2:     m_p ← len(h)
 3:     if m_p + 1 is even then
 4:         d ← (m_p + 1)/2
 5:         r ← d
 6:     else
 7:         d ← m_p/2
 8:         r ← d + 1
 9:     M ← LLL-reduction of matrix M_2 described above.
10:     β ← (M_{0,0}, ..., M_{0,d−1})
11:     α ← β/2^{ℓ−1}
12:     P ← α_0 + α_1 Z + ... α_{d−1} Z^{d−1}                ▷ P is a polynomial
13:     return P
14: procedure AttackSimpleStern(h, N, ℓ, m_p)
15:     m ← len(h)
16:     t = ⌊m_p/2⌋
17:     h_1 = (h_0, ..., h_{m_p−1})
18:     P_1 ← FindPolynomial(h_1, N, ℓ)
19:     h_2 = (h_t, ..., h_{t+m_p−1})
20:     P_2 ← FindPolynomial(h_2, N, ℓ)
21:     P ← GCD(ℤ/Nℤ, P_1, P_2)
22:     if deg(P) == 1 then                                  ▷ P = γ_0 + γ_1 Z
23:         a = −γ_0/γ_1 mod N
24:         x_0 ← AttackFrieze(h, a, N, m, ℓ))
25:         return x_0, a
```

As we only use an instance of the LLL algorithm on a matrix($(r + d) \times (r + d)$), the time complexity of this algorithm is polynomial on m_p.

The first limit in this attack is eq.(2.3). We need $|\sum_{i=0}^{d-1} \mu_i \delta_{i+j}| < N$, in other words $d2^{n/d}2^{l-1} < 2^n$ which is no longer possible if ℓ is too close to n. When this inequality is not satisfied any more, the vector v is not the shortest neither even particularly short in Λ_2.

The second limit is given by the bound $\ell < n(1 - 2/d) + 2 - \log_2(r^2 + 1)$. For a given n, we can compute the maximum of the right term. The results are presented in the following table.

n	32	64	1024
$\ell \leq$	22	52	1004

All these calculus allow us to predict if the constructed polynomial will have a as a root modulo

N. We have no heuristic for the second part of the algorithm but we found heuristically that is was hard to constantly obtain a polynomial of degree exactly one.

Experimental results

For a given n and r we search for the greater ℓ such that the probability of success of finding a polynomial P such that $P(a) \equiv 0 \bmod N$ is above 50%.

m	4	5	6	7	16
$n = 32$					
ℓ (th.) $\leq$	5	9	12	13	20
ℓ (exp.) $\leq$	5	11	13	16	24
time	$0.004s$	$0.006s$	$0.007s$	$0.01s$	$0.04s$
$n = 64$					
ℓ (th.) $\leq$	10	20	25	29	45
ℓ (exp.) $\leq$	10	21	26	32	48
time	$0.005s$	$0.007s$	$0.008s$	$0.01s$	$0.05s$
$n = 1024$					
ℓ (th.) $\leq$	170	340	425	509	778
ℓ (exp.) $\leq$	170	341	426	512	781
time	$0.006s$	$0.008s$	$0.01s$	$0.013s$	$0.07s$

In the case where N is prime, for a given n and r we search for the greater ℓ such that the probability of success of retrieving the multiplier a is above 20% (we lower the bar as the second part of the algorithm brings a lot of failure).

m_p	4	5	6	7	16
m	6	8	9	11	24
$n = 32$					
ℓ (th.) $\leq$	5	9	12	13	20
ℓ (exp.) $\leq$	6	10	12	15	24
time	$0.01s$	$0.014s$	$0.018s$	$0.023s$	$0.08s$
$n = 64$					
ℓ (th.) $\leq$	10	20	25	29	45
ℓ (exp.) $\leq$	11	21	25	32	48
time	$0.011s$	$0.014s$	$0.018s$	$0.024s$	$0.13s$
$n = 1024$					
ℓ (th.) $\leq$	170	340	425	509	778
ℓ (exp.) $\leq$	172	341	425	511	782
time	$0.012s$	$0.014s$	$0.021s$	$0.035s$	$0.14s$

Alternate ending when $N = 2^n$

If we know a polynomial P such that $P(a) = 0 \bmod 2^n$ and if P is not degenerate, we can hope to find a using Hensel lifting.

In the case where $N = 2^n$, for a given n and r we search for the greater ℓ such that the probability of success of retrieving the multiplier a is above 50%.

Algorithm 5 Seed retriever using Stern method when $N = 2^n$

```
1: procedure HenselLifting(P, n)
2:     roots ← []
3:     if P(0) == 0 mod 2 then
4:         roots ← roots + [0]                  ▷ the + here represents concatenation
5:     if P(1) == 0 mod 2 then
6:         roots ← roots + [1]
7:     for i ∈ {2, ..., n} do
8:         newroots ← []
9:         for z ∈ roots do
10:            if P(z) == 0 mod 2^i then
11:                newroots ← newroots + [z]
12:            if P(z + 2^(i−1)) == 0 mod 2^i then
13:                newroots ← newroots + [z + 2^(i−1)]
14:        roots ← newroots
15:        if len(roots) > 100 then
16:            return          ▷ we abort if the number of roots is not manageable
17:    return roots
18: procedure AttackSimpleSternAlt(h, 2^n, m, ℓ)
19:    P ← FindPolynomial(h, 2^n, m, ℓ)
20:    roots ← HenselLifting(P, n)
21:    for a ∈ roots do
22:        x_0 ← AttackFrieze(h, a, N, m, ℓ)
23:        if CheckConsistency(x_0, a, N, h) then
24:            return x_0
```

m	4	5	6	7
$n = 32$				
ℓ (th.) $\leq$	5	9	12	13
ℓ (exp.) $\leq$	5	10	13	26
time	$0.013s$	$0.018s$	$0.033s$	$0.026s$
$n = 64$				
ℓ (th.) $\leq$	10	20	25	29
ℓ (exp.) $\leq$	10	21	26	32
time	$0.010s$	$0.018s$	$0.030s$	$0.040s$
$n = 1024$				
ℓ (th.) $\leq$	170	340	425	509
ℓ (exp.) $\leq$	170	341	426	512
time	$0.027s$	$0.046s$	$0.053s$	$0.043s$

2.2.3 Knuth Attack on the Knuth generator

This attack is a bit aside because it focuses on a particular LCG and does not use lattice technique. Even if this attack is not particularly effective it seemed important to present it as it is one of the oldest against the LCG. It was presented by Knuth in 1985 in [36].

We consider the *Knuth Generator* of seed x_0. At step i it computes : $x_{i+1} \equiv ax_i + c \bmod 2^k$ and outputs $y_{i+1} = x_{i+1} \gg \ell$. The parameters a and c are secret and satisfy $a \equiv 1 \bmod 4$ and $c \equiv 1 \bmod 2$.

For a integer t we consider $z_n^{(t)} = x_{n+2^t} - x_n \bmod 2^k$. We obtain two properties:

$$z_{n+1}^{(t)} \equiv az_n^{(t)} \bmod 2^k \tag{2.4}$$

$$z_n^{(t)} \text{ is an odd multiple of } 2^t \tag{2.5}$$

Now we call $x_n^{(t)}$ the t-th bit from the right of x_n and consider the following lemma.

Lemma 1. *For each t in $\{1, \dots, k-2\}$, there exists a unique b_t such that for any n,*

$$x_n^{(t)} \equiv x_{n+2^{t-1}}^{(t+1)} - x_n^{(\ell+1)} + b_t \bmod 2. \tag{2.6}$$

Proof. Because of 2.5, there exists b_t such that $z_0^{(t)} \equiv b_t 2^t + 2^{t-1} \bmod 2^{t+1}$. As $z_{n+1}^{(t)} \equiv az_n^{(t)} \bmod 2^k$ and $a \equiv 1 \bmod 4$, for every n, $z_n^{(t)} \equiv b_t 2^t + 2^{t-1} \bmod 2^{t+1}$, always with the same b_t. Then we notice that $z_n^{(t-1)} + x_n \equiv x_{n+2^{t-1}} \bmod 2^k$, hence the $(t+1)$-th bit of $z_n^{(t-1)} + x_n$ is equal to $x_{n+2^{t-1}}^{(t)}$. It gives the expected result.

□

Thanks to this lemma we can easily see the trajectory of the attack. We start by guessing b_ℓ, and as we know 2^ℓ outputs we can guess all the x_n^ℓ with the equation 2.6. Then we guess $b_{\ell-1}$ and derive all the $x_n^{\ell-1}$ and so on. This method is not efficient as its time complexity is in $O(2^\ell)$.

2.3 Attacks when both the multiplier and the modulus are unknown

The internal states of the generator satisfy the following equation

$$x_{i+1} \equiv ax_i \bmod N$$

where both the multiplier a and the modulus N are secret.

2.3.1 Presentation of the attack

This attack was presented by Stern in 1987 in and improved by Contini and Shparlinski in 2005 [17].

Step 1: Constructing polynomials Let r and d be again two integers, we want to construct several polynomials P_j of degree $d-1$ such that $P_j(a) \equiv 0 \bmod N$.

Let $\mathbf{x}_k$ be $(x_k, \ldots, x_{k+r-1})$. As above, we want a linear combination of the $\mathbf{x}_i$ that sums to zero, but this time it has to be on the integers as we do not know the modulus N. We are searching for $(\mu_0, \ldots, \mu_{d-1})$ such that

$$\sum_{i=0}^{d-1} \mu_i \mathbf{x}_i = 0. \tag{2.7}$$

As $x_i = a^i x_0$, these μ_i would give us a polynomial P_j such that $P_j(a) \equiv 0 \bmod N$.

We do not know the $\mathbf{x}_i$, so we cannot find these μ_i so easily. Instead, we will search for $\boldsymbol{\mu} = (\mu_0, \ldots, \mu_{d-1})$ such that

$$\sum_{i=0}^{d-1} \mu_i \mathbf{y}_i = 0. \tag{2.8}$$

where $\mathbf{y}_k = (y_k, \ldots, y_{k+r-1})$

We will present conditions that force solutions of eq.(2.8) to also satisfy eq.(2.7).

To find a small solution of eq(2.8), we apply the LLL-algorithm on the following matrix

$$M = \left(\begin{array}{cccc|c} 1 & & & & k\mathbf{y}_0 \\ & 1 & & & k\mathbf{y}_1 \\ & & \ddots & & \vdots \\ & & & 1 & k\mathbf{y}_{d-1} \end{array} \right)$$

where k is an integer parameter to define.

If the small solution of eq(2.8) is also a solution of eq(2.7) then it gives a polynomial P_j such that $P_j(a) \equiv 0 \bmod N$.

Step 2: Retrieving the modulus N Here we present the alternate version of step 2, presented by Contini and Shparlinski [17].

If $P_i(a) \equiv 0 \bmod N$ and $P_j(a) \equiv 0 \bmod N$ then N divides the resultant of P_i and P_j. The algorithm will simply apply the first step several time to obtain several polynomials. Then it will compute resultants and GCDs to obtain N or a small multiple.

2.3.2 Theoretical choice of the parameters

In the article we find the following proposition.

Proposition 3. *There exists a solution to eq(2.8) such that its coefficients are bounded by B where*

$$B = 2^{(n-l+\log(d)-1)r/(d-r)}.$$

A slightly bigger solution $\boldsymbol{\mu}$ can be computed as the first line of the LLL-reduction of M where $k = \lceil \sqrt{d} 2^{(d-1)/2} B \rceil$. This solution satisfies $\|\boldsymbol{\mu}\|_\infty < k$.

Heuristically the solution given by the proposition should also satisfy eq.(2.7) when

$$\ell < (r-1)n/r \text{ and } d \simeq \sqrt{2nr}$$

Algorithm 6 Description of Stern Algorithm

```
1: procedure STEP1(y, r, d, ℓ)
2:     B ← 2^((n−l+log(d)−1)r/(d−r))
3:     k ← ⌈√d 2^((d−1)/2) B⌉
4:     M ← LLL-reduction of the matrix M described above
5:     μ ← (M_{0,0}, ..., M_{0,r−1})
6:     return μ
7: procedure ATTACKSTERN(y, r, ℓ)
8:     d ← ⌊√(2(n − ℓ)r)⌋
9:     μ ← STEP1((y_0, ..., y_{d+r−1}), r, d, ℓ)
10:    P_1 ← μ_0 + μ_1 Z + μ_2 Z^2 + ··· + μ_{r−1} Z^{r−1}
11:    γ ← STEP1((y_1, ..., y_{d+r}), r, d, ℓ)
12:    P_2 ← γ_0 + γ_1 Z + γ_2 Z^2 + ··· + γ_{r−1} Z^{r−1}
13:    η ← STEP1((y_2, ..., y_{d+r+1}), r, d, ℓ)
14:    P_3 ← η_0 + η_1 Z + η_2 Z^2 + ··· + η_{r−1} Z^{r−1}
15:    R_12 ← RESULTANT(P_1, P_2)
16:    R_23 ← RESULTANT(P_2, P_3)
17:    R_13 ← RESULTANT(P_1, P_3)
18:    Ñ ← GCD(R_12, R_23, R_13)
19:    P ← GCD(ℤ/Ñℤ, P1, P2)
20:    P ← GCD(ℤ/Ñℤ, P, P3)
21:    if deg(P) == 1 then                                ▷ P = γ_0 + γ_1 Z
22:        a = −γ_0/γ_1 mod Ñ
23:        x_0 ← ATTACKFRIEZE(h, a, Ñ, m, ℓ))
24:        return x_0
```

Complexity and limits

The algorithm applies three times the LLL algorithm on a matrix of size $d \times 2r$, three resultants on polynomials of degree r and GCD on theses resultants. As d is close to $\sqrt{2n}$, we can conclude that the time complexity of this algorithm is polynomial in n.

Experimental results

For a given n and r we search for the greater ℓ such that the probability of success of retrieving the multiplier a and N is above 50%.

- $n = 32$

m	23	28	33	38
ℓ (th.) $<$	21	22	25	26
ℓ (exp.) $\leq$	17	19	21	22
time	$0.10s$	$0.14s$	$0.17s$	$0.21s$

- $n = 64$

m	29	35	41	46
ℓ (th.) $<$	42	47	51	53
ℓ (exp.) $\leq$	37	43	46	48
time	$0.20s$	$0.27s$	$0.34s$	$0.39s$

- $n = 1024$

m	56	60	64	66
ℓ (th.) $<$	682	768	819	853
ℓ (exp.) $\leq$	656	740	791	834
time	$5.5s$	$7.1s$	$6.8s$	$5.8s$

The results are below the heuristic. Even if we reduce the problem to finding polynomials satisfying $P(a) \equiv 0 \bmod N$.

2.4 Variants of the Linear Congruential Generator

2.4.1 One output over two

The internal states of the generator satisfy the following equation

$$x_{i+1} \equiv ax_i + c \bmod N$$

but the outputs are only issued from one internal state over two. We construct intermediate states v_i given by

$$v_i = x_{2i}$$

and the outputs y_i are given by $v_i \equiv y_i 2^\ell + \delta_i$.

When we write v_{i+1} as a function of v_i we obtain

$$\begin{aligned} v_{i+1} &\equiv x_{2i+2} \bmod N \\ &\equiv ax_{2i+1} + c \bmod N \\ &\equiv a(ax_{2i} + c) + c \bmod N \\ &\equiv a'v_i + c' \bmod N \end{aligned}$$

where $a' = a^2$ and $c' = (a+1)c$. Using only one output over a fixed number does not increase the difficulty of retrieving the seed. We face the same problem with the same parameters but different constants.

2.4.2 Upper bits truncated

The internal states of the generator still satisfy the following equation $x_{i+1} \equiv ax_i + c \bmod N$ and $x_i = \delta_i 2^{n-\ell} + y_i$.

If N is a **power of 2**, the outputs directly satisfy $y_{i+1} \equiv ay_i + c \bmod 2^{n-\ell}$. Predicting the generator becomes trivial but the seed can never be recovered.

If N is **known and odd**, then we can invert $2^{n-\ell} \bmod N$ and we call this inverse α. Then

$$\alpha x_i \equiv \alpha y_i + \delta_i \bmod N$$

where $|\delta_i| \leq 2^\ell$. If we call $v_i = \alpha x_i$, then

$$v_{i+1} \equiv a z_i + c' \bmod N$$

where $c' = \alpha c$ and

$$v_i = y'_i + \delta_i$$

where $y'_i = \alpha y_i$. Our problem is very similar to the classical LCG and most of the attacks work without modification. The only detail is that y'_i might not be a multiple of 2^ℓ but we can discard the least significant bit to retrieve the same exact situation.

If N is **known and even** we split it in two $N = 2^p Q$ where Q is odd. We consider on one side the generator $\text{mod} 2^{min(p,\ell)}$ and on the other side the generator $\text{mod} Q$. We predict outputs for these two sequences and reconstruct the outputs of the original generator thanks to the Chinese Reminder Theorem. This problem with parameters N, ℓ is as hard as predicting a classical LCG of parameters Q, ℓ with $\log_2(Q) < \log_2(N)$.

If N is **unknown**, the problem seems far more complex.

Part II

Reducing Pseudo Random Number Generators to Linear Congruential Generators

Chapter 3

The Permuted Congruential Generator

This chapter is largely inspired by the article *Practical seed-recovery for the PCG Pseudo-Random Number Generator* co-written with Charles Bouillaguet and Julia Sauvage and presented at FSE 2020 [15]

The *Permuted Congruential Generator* (PCG) is the default pseudo-random number generator in the popular NumPy [63] scientific computing package for Python. It essentially consists in applying a non-linear filtering function on top of a LCG. The resulting combination is fast and passes current statistical test suites. The PCG family contains many members, but we focus on the strongest one, named either PCG64 or PCG-XSL-RR. The internal state of the PCG64 generator is made of a 128-bit "state" and a 128-bit "increment", whose intended use is to provide several pseudo-random streams with the same seed. A default increment is provided in case the end-user just want one pseudo-random stream with a single 128-bit seed.

We describe an algorithm that reconstructs the full internal state of the strongest member of the PCG family. This allows to predict the pseudo-random stream deterministically and clock the generator backwards. The original seeds can also easily be reconstructed. The state reconstruction algorithm is practical and we have executed it in practice. It follows that predicting the output of the PCG should be considered practically feasible.

While the PCG pseudo-random generator is not meant as a cryptographic primitive, obtaining an actual prediction algorithm requires the use of cryptanalytic techniques. Making it practical requires in addition a non-trivial implementation effort.

Our algorithm reconstructs the internal state using 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.

Notation.

- *If* $\mathbf{x} = (x_0 \dots x_{n-1}) \in \{0,1\}^n$ *is an* n*-bit string, then* $\mathbf{x}[i{:}j]$ *denotes the bit string* $(x_i x_{i+1} \dots x_{j-2} x_{j-1})$ *(this is the "slice notation" used in Python)*
- *If* $\mathbf{U}$ *is a vector or a sequence, then* U_i *is the* i*-th element and we will use capital letters for the integers that we will consider as bit strings*

- *Modular addition is denoted + (or ⊞ to make it even more explicit).*
- *In the rest of this chapter, we often perform arithmetic operations on integers where only some bits are known. This leads to generation of unknown carries. If a, b are integers modulo 2^{128} and $0 \leq i < j < 128$, then there is a* carry $0 \leq \gamma \leq 1$ *(resp. a* borrow $0 \leq \beta \leq 1$*) such that:*

$$(a \boxplus b)[i{:}j] = a[i{:}j] \boxplus b[i{:}j] \boxplus \gamma, \tag{3.1}$$

$$(a \boxminus b)[i{:}j] = a[i{:}j] \boxminus b[i{:}j] \boxminus \beta. \tag{3.2}$$

3.1 Presentation of the Generator

We describe the PCG64, a non-cryptographic pseudo-random number generator (a.k.a. PCG-XSL-RR in the designer's terminology).

PCG64 has an internal state of 128-bit, which operates as a linear congruential generator modulo 2^{128}. More precisely:

$$S_{i+1} = aS_i + c \bmod 2^{128},$$

where the multiplier a is a fixed 126-bit constant. The first initial state S_0 is the seed of the generator. The increment c may be specified by the user of the PRNG to produce different output streams with the same seed (just as the IV acts in a stream cipher). If no value of c is specified, then a default increment is provided. Note that c must be odd. The default values are:

$$a = 47026247687942121848144207491837523525 \quad \text{(fixed)}$$

$$c = 117397592171526113268558934119004209487 \quad \text{(default value, user-definable)}$$

Each time the PRNG is clocked, 64 output bits are extracted from the internal state using a non-linear function that makes use of data-dependent rotations. The six most significant bits of the internal state encode a number $0 \leq r < 64$. The two 64-bit halves of the internal state are XORed together, and this 64-bit result is rotated right by r positions.

The successive 64-bit outputs of the generator are $X_0, X_1, \ldots$ where:

$$X_i = (\underbrace{S_i[0{:}64] \oplus S_i[64{:}128]}_{Y_i}) \ggg \underbrace{S_i[122{:}128]}_{r_i}. \tag{3.3}$$

For the sake of convenience, we denote by Y_i the XOR of the two halves of the state (before the rotation) and by r_i the number of shifts of the "i-th rotation".

Fig. 3.1 summaries the process. The overall design strategy is similar to that of a filtered LFSR: the successive states of a weak internal generator with a strong algebraic structure are "filtered" by a non-linear function.

Updating the internal state requires a $128 \times 128 \rightarrow 128$ multiplication operation. In fact, this can be done with three $64 \times 64 \rightarrow 128$ multiplications and two 64-bit additions. High-end desktop CPUs all implement these operations in hardware, so the generator is quite fast on these platforms.

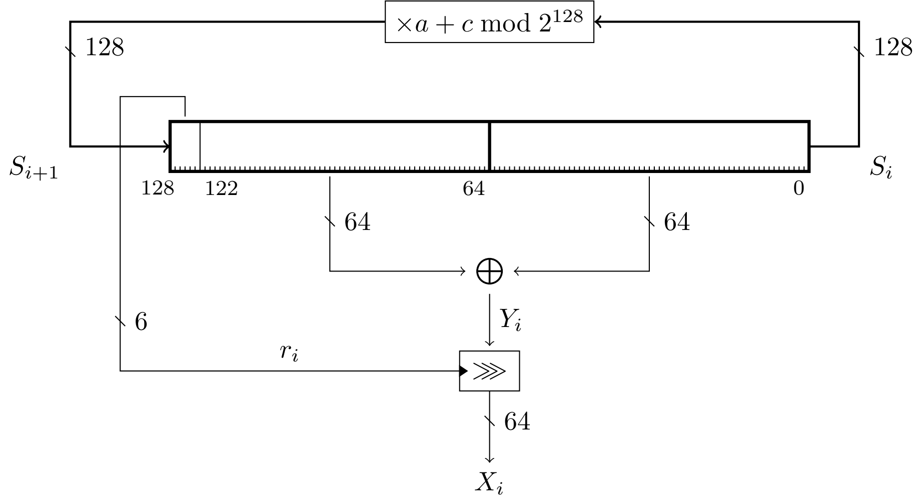


Figure 3.1: PCG64: Internal state update and output process.

3.2 Dealing with a noisy truncated Linear Congruential Generator

Given an integer k, a fixed multiplier a and a "seed" x, we define the sequence:

$$U_0 = x$$
$$U_{i+1} \equiv a \times U_i \bmod 2^k.$$

The vector $\mathbf{U}$ forms the n successive states of a LCG.

Let $T_i = U_i[k-t{:}k]$ denote the top t bits of U_i and Δ_i denote the lower $k-t$ bits, then $U_i = T_i 2^{k-t} + \Delta_i$ and $0 \leq \Delta_i < 2^{k-t}$. We consider $\boldsymbol{\varepsilon}$ an arbitrary "noise vector" such that $\varepsilon_i \in \{-1, 0, 1\}$. Finally, we set $\widetilde{T}_i = T_i + \varepsilon_i \bmod 2^t$.

Lemma 2. *There exists* $\mathbf{U}'$ *such that* $U'_i \equiv U_i \bmod 2^k$ *and* $\left\|\mathbf{U}' - 2^{k-t}\widetilde{\mathbf{T}}\right\|_2 \leq \sqrt{n}2^{k-t+1}$.

Proof.

- If $\widetilde{T}_i = T_i + \varepsilon_i$ (without modulo), then we have:

$$\begin{aligned}|U_i - 2^{k-t}\widetilde{T}_i| &= |T_i 2^{k-t} + \Delta_i - T_i 2^{k-t} - 2^{k-t}\varepsilon_i| \\ &= |\Delta_i - 2^{k-t}\varepsilon_i| \\ &< 2^{k-t+1}\end{aligned}$$

- If $\widetilde{T}_i = T_i + \varepsilon_i + 2^t$, we fix $U'_i = U_i + 2^k$ and we have:

$$\begin{aligned}|U'_i + 2^{k-t}\widetilde{T}_i| &= |U_i + 2^k - 2^{k-t}(T_i + \varepsilon_i) - 2^{k-t}2^t| \\ &= |\Delta_i - 2^{k-t}\varepsilon_i| \\ &< 2^{k-t+1}\end{aligned}$$

- If $\widetilde{T}_i = T_i + \varepsilon_i - 2^t$, we fix $U'_i = U_i - 2^k$ and we have:

$$\begin{aligned} |U'_i - 2^{k-t}\widetilde{T}_i|_2 &= |U_i - 2^k - 2^{k-t}(T_i + \varepsilon_i) + 2^{k-t}2^t| \\ &= |\Delta_i - 2^{k-t}\varepsilon_i| \\ &< 2^{k-t+1} \end{aligned}$$

□

This means this noisy LCG where we truncate $k-t$ bits can be seen and attacked as a classical LCG where we truncate $k-t+1$ bits.

In section 3.4.3, we will be facing the problem of reconstructing a geometric sequence modulo 2^{128} given arbitrarily many (noisy versions of the) most-significant 6 bits of successive elements of the sequence. To do so we will use an exact CVP solver on the lattice Λ_n spanned by the rows of the following matrix

$$G_{n,k} = \begin{pmatrix} 1 & a & a^2 & \dots & a^{n-1} \\ & 2^k & & & \\ & & 2^k & & \\ & & & \ddots & \\ & & & & 2^k \end{pmatrix}$$

already presented in section 2.1.1, with $k = 128$ and $t = 6$. We are searching for a parameter n such that $\mathbf{U}'$ described above is the closest vector to $2^{122}\widetilde{\mathbf{T}}$ in Λ_n.

We said earlier that our problem could be seen as attacking a classical LCG missing $k+t+1$ bits, so we might want to use the results of section 2.1. If we write $U_i = 2^{k-t}\widetilde{T}_i + \delta_i$, then $\delta_i < 2^{k-t+1}$ (in the case $\varepsilon_i = 1$) and we cannot use the trick where we recenter the discarded bits around zero as δ_i might not be positive (in the case $\varepsilon_i = -1$). Because of that the equation 2.1 becomes $(k-t+1)+1 < k(n-1)/n$. It gives $n = 32$. But this reasoning was heuristic (as we used the Gaussian heuristic). If we redo the calculus that gives the equation but we keep λ_1 as such, we obtain the condition $2\sqrt{n}2^{k-t+1} < \lambda_1(\Lambda_n 2.1)$.

Starting from $n = \lceil 122/6 \rceil$, we computed the length of the shortest vector of the lattice spanned by $G_{n,128}$ for each successive n until the condition holds. To solve these SVP we used the (almost) off-the-shelf `G6K` library [2], which gave results very quickly by sieving. `fplll` [61] was too slow above dimension 50, in the default settings.

After this computation, we found that the minimal possible n is 63: with $n = 63$, the shortest vector of Λ_n has length greater than $2^{127.02}$, which is high enough. This vector can be obtained by bootstrapping the geometric sequence with

$$U_0 = 121442528758503454790150022052419873 63.$$

It follows that when $n \geq 63$, $k = 128$ and $t = 6$, any CVP oracle will return a vector congruent to the original $\mathbf{U}$ when given $\widetilde{\mathbf{T}}$.

3.2.1 Reconstruction in Low Dimension Using Babai's Rounding

In sections 3.3 and 3.4.1 we will need to reconstruct billions of noisy truncated geometric series modulo 2^{64} with very few terms, of which a large fraction of most-significant bits are known. In

this setting, the CVP problem becomes much easier. This enables us to use faster and more *ad hoc* methods, such as Babai's rounding algorithm.

Denote again by Λ_n the n-dimensional lattice spanned by the rows of $G_{n,64}$, and let H denote the LLL-reduction of $G_{n,64}$. The same lattice is also spanned by the rows of H. For instance, with $n = 3$:

$$H = \begin{pmatrix} -1241281756092 & 3827459685972 & -728312298332 \\ -5001120657083 & -2117155768935 & 5479732607037 \\ 8655886039732 & 3303731088004 & 6319848582548 \end{pmatrix}$$

As we want to retrieve $\mathbf{U}'$ as defined in the previous subsection, we will need it to satisfy two conditions

- The vector $\mathbf{U}'$ is the closest vector to $2^{k-t}\widetilde{\mathbf{T}}$ in Λ_n. By the previous section it means

$$2\sqrt{n}2^{k-t+1} < \lambda_1(\Lambda_n)$$

- The Babai rounding method should return the closest vector to $2^{k-t}\widetilde{\mathbf{T}}$ in Λ_n (which should be $\mathbf{U}'$ by the first condition), hence

$$(1 + \text{cond}(H))\|\mathbf{U}' - 2^{k-t}\widetilde{\mathbf{T}}\|_2 < \lambda_1(\Lambda)$$

  as seen in section 1.4.

We have seen earlier that $\|\mathbf{U}' - 2^{k-t}\widetilde{\mathbf{T}}\|_2 < \sqrt{n}2^{k-t+1}$ and that the condition number of a matrix is always greater or equal to one. We only have one condition left which is:

$$(1 + \text{cond}(H))\sqrt{n}2^{k-t+1} < \lambda_1(\Lambda).$$

n	$\|\|\|H\|\|\| \times \|\|\|H^{-1}\|\|\|$	$\lambda_1(\Lambda)$	minimum t	$\left(1 + \|\|\|H\|\|\| \times \|\|\|H^{-1}\|\|\|\right)\sqrt{n}2^{64-t+1}$
3	2.87	$4.09e^{12} \simeq 2^{41.9}$	26	$3.69e^{12}$
4	2.06	$2.44e^{14} \simeq 2^{47.8}$	20	$2.15e^{14}$
5	3.77	$1.72e^{15} \simeq 2^{50.6}$	18	$1.5e^{15}$
6	2.69	$1.03e^{16} \simeq 2^{53.2}$	15	$1.02e^{16}$

Table 3.1: minimal t needed for a given n

When t is greater than the values given in table 3.1, then Babai's rounding technique will always return the closest vector, and will allow us to reconstruct a truncated geometric serie.

3.3 State Reconstruction for PCG64 With Known Increment

We first consider the easier case where the increment c is known — recall that a default value is specified in case the user of the pseudo-random generator does not want to provide one.

In this case, reconstructing the 128-bit internal state S_i of the generator is sufficient to produce the pseudo-random flow with 100% accuracy (the generator can also be clocked backwards if necessary, so that the seed can be easily reconstructed). We therefore focus on reconstructing S_0 (the seed) from $X_0, X_1, X_2, \ldots$. A very simple strategy could be the following:

1. Guess the 64 upper bits of S_0 (this includes the rotation).
2. Compute the missing 64 lower bits using (3.3), with:

$$S_0[0{:}64] = S_0[64{:}128] \oplus (X_0 \lll S[122{:}128]).$$

3. Compute S_1 then extract X_1; if X_1 is correct, then output S_0.

This "baseline" procedure requires 2^{64} iterations of a loop that does a dozen arithmetic operations; it always outputs the correct value of S_0, and may output a few other ones (they can be easily discarded by checking X_2). An improved "guess-and-determine" state reconstruction algorithm is possible, which essentially amounts to expose a truncated version of the underlying linear congruential generator, and attack it using the tools exposed in chapter 2 and section 3.2. This is possible by combining the following ingredients:

- The underlying linear congruential generator uses a power-of-two modulus, therefore the ℓ low-order bits of S_{i+1} are entirely determined by the ℓ low-order bits of S_i. More precisely, we have:

$$S_{i+1} = aS_i + c \bmod 2^{\ell}, \qquad \text{for all } 0 \leq \ell \leq 128 \tag{3.4}$$

  Therefore, guessing the least-significant bits of S_0 yields a "long-term advantage" that holds for all subsequent states.

- Guessing a 6-bit rotation r_i gives access to Y_i (the XOR of the two halves of the internal state). Thus, if a part of the state is known, then this transfers existing knowledge to the other half.

In figure 3.2, we see that guessing $S_0[0{:}\ell]$ and a few 6-bit rotations r_i give access to $S_i[58{:}64+\ell]$ for the corresponding states. Therefore, looking at $S_i[\ell{:}64+\ell]$, we are facing a truncated linear congruential generator on 64 bits, where we have access to the $6+\ell$ most-significant bits of each state (denoted by T), for a few consecutive states. This is sufficient to reconstruct entirely the successive states of this truncated linear congruential generator. This reveals $S_0[\ell{:}64+\ell]$, and using (3.3) the entire S_0 can be reconstructed. The precise details follow.

We consider the sequence of internal states $S = (S_0, S_1, \dots) = \mathrm{LCG}_{128}(S_0, c)$. We will guess the ℓ least-significant bits of S_0, therefore let us assume that their value is known and denote it by w. We define $\mathbf{S}' = \mathrm{LCG}_{128}(S_0 - w, 0)$ and $\mathbf{K} = \mathrm{LCG}_{128}(w, c)$ — this is known. As the LCG is *linear*, we have $\mathbf{S}' = \mathbf{S} - \mathbf{K}$. The point is that the elements of $\mathbf{S}'$ follow a geometric progression of common ratio a; in addition, the ℓ least significant bits of each components are equal to zero. It follows that if we fix $\mathbf{U} = \mathbf{S}'[\ell{:}64+\ell]$, $\mathbf{U}$ also follows a geometric progression of common ratio a, this time modulo 2^{64}. The crux of the reconstruction algorithm is to find $\mathbf{U}$.

As we know $\mathbf{K}$, for each guessed rotation r_i we have access to $S_i[58{:}64+\ell]$ (named T_i, in yellow on the figure). Then

$$\begin{aligned}
\mathbf{U}[58-\ell{:}64] &= \mathbf{S}'[58{:}64+\ell] \\
&= (\mathbf{S} \boxminus \mathbf{K})[58{:}64+\ell] \\
&= \mathbf{S}[58{:}64+\ell] \boxminus \mathbf{K}[58{:}64+\ell] \boxminus \mathbf{B}
\end{aligned}$$

where $\mathbf{B}$ is an unknown vector of *borrows*, whose components are either 0 or 1, by (3.2).

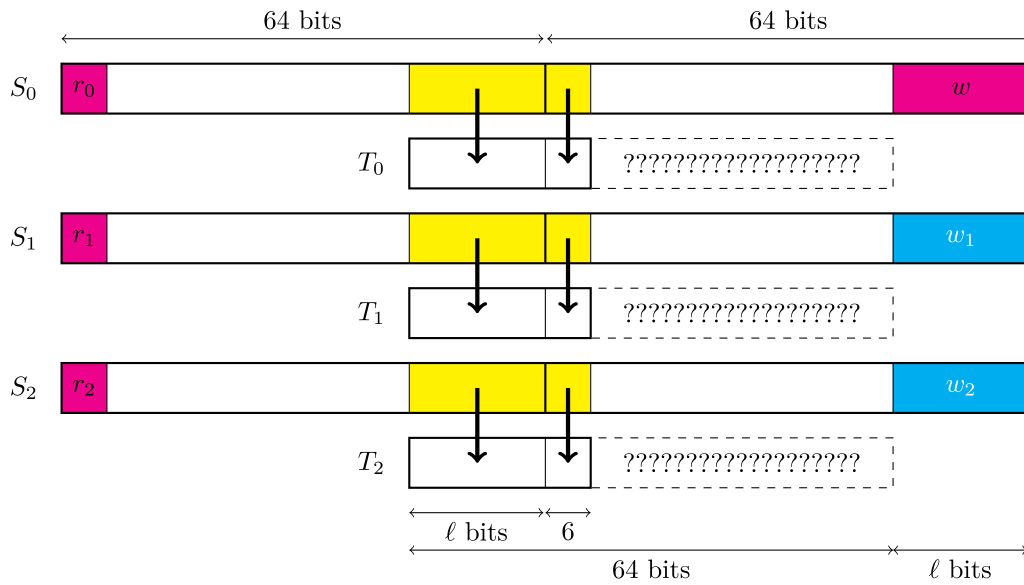


Figure 3.2: A guess-and-determine algorithm to reconstruct the first internal state S_0. Magenta bits are guessed; cyan bits are obtained using the linear congruence relation (3.4) modulo 2^ℓ; yellow bits are obtained from the output and the guessed rotations using (3.3).

We can compute $\widetilde{\mathbf{T'}} = \mathbf{S}[58{:}64+\ell] \boxminus \mathbf{K}[58{:}64+\ell]$, and clearly $\widetilde{\mathbf{T'}} = \mathbf{U}[58-\ell{:}64] \boxplus \mathbf{B}$. We are thus in the context of the problem discussed in section 3.2, namely reconstructing a geometric sequence given $t = 6 + \ell$ (noisy) most-significant bits. The "noise" is the unknown vector $\mathbf{B}$ of borrows.

We will guess n rotations and ℓ least-significant bits of the state, for a total of $2^{6n+\ell}$ guessed bits. Table 3.1 gives a lower-bound on $t = 6 + \ell$ given n, and we see that the total number of guessed bits reaches a minimum of 38 when $n = 3$ and $\ell = 20$. Therefore, success is guaranteed if we guess $\ell = 20$ low-order bits of the state and three consecutive rotations.

The algorithm that reconstructs the internal state of the PCG64 generator with known increment proceeds as shown in algorithm 7.

The procedure is completely practical. More details are given in section 3.5. Let us just mention that the procedure often works (twice faster) with $\ell = 19$ or even four times faster with $\ell = 18$ (with a reduced success probablity).

Algorithm 7 State reconstruction Algorithm (case where c is known)

1: **procedure** RECONSTRUCTSTATE$_\ell(X_0, X_1, X_2)$
2:     *// Statement involving j must be repeated for $j = 0, 1, 2$.*
3:     $H \leftarrow$ LLL reduction of $G_{3,64}$
4:     $\ell \leftarrow 20$
5:     **for** $0 \le w < 2^\ell$ **do** ▷ Guess least-significant bits of S_0
6:         $K_j \leftarrow a^j w + c(a^j - 1)(a-1)^{-1} \bmod 2^{128}$ ▷ Known part
7:         **for** $0 \le r_0, r_1, r_2 < 64$ **do** ▷ Guess rotations
8:             $Y_j \leftarrow X_j \lll r_j$ ▷ Undo rotations
9:             $T_j \leftarrow (r_j \oplus Y_j[58{:}64]) + 64 \cdot (K_j \oplus Y_j)\,[0{:}\ell]$ ▷ Truncated LCG output
10:             $\widetilde{T}'_j \leftarrow T_j \boxminus K_j[58{:}64+\ell]$ ▷ Truncated geometric series on $6+\ell$ bits
11:             $(U_0, U_1, U_2) \leftarrow \left\lfloor 2^{58-\ell} \cdot (\widetilde{T}'_0, \widetilde{T}'_1, \widetilde{T}'_2) \cdot H^{-1} \right\rceil \cdot H$ ▷ CVP (Babai rounding)
12:             $S_0[0{:}64] \leftarrow K_0[0{:}64] + 2^\ell \cdot U_0[0{:}64-\ell]$ ▷ Reconstruct S_0
13:             $S_0[64{:}128] \leftarrow S_0[0{:}64] \oplus Y_0$
14:             $S_1 \leftarrow aS_0 + c$ ▷ Recompute X_1
15:             $\widehat{Y}_1 = S_1[0{:}64] \oplus S_1[64{:}128]$
16:             **if** $\widehat{Y}_1 = Y_1$ **then** ▷ Check consistency
17:                 output S_0 as a candidate internal state.

3.4 State Reconstruction for PCG64 With Secret Increment

The algorithm of section 3.3 does not apply directly to the general case where the value of c is unknown. A "baseline" procedure would consist in guessing $S_0[64{:}128]$ and $S_1[64{:}128]$; using eq. (3.3), this would reveal S_0 and S_1; from there, the increment c is easy to obtain, and every secret information has been reconstructed. This would take 2^{128} iterations of a very simple procedure, which is completely infeasible.

Set $\Delta S_i = S_{i+1} \boxminus S_i$; it is easily checked that ΔS_i is a geometric progression of common ratio a. Therefore, reconstructing both S_0 and ΔS_0 is sufficient to compute all subsequent states (and recover the unknown increment c). The global "guess-and-determine" strategy is essentially the same as before: gaining access to a truncated version of ΔS_i, solving a small SVP instance,

reconstructing ΔS_0, then checking consistency.

Let us set:

$$\nabla S_i \stackrel{def}{=} S_i - S_0 \equiv \sum_{j=0}^{i-1} \Delta S_j \equiv \Delta S_0 \cdot \sum_{j=0}^{i-1} a^j \equiv \Delta S_0 \frac{a^i - 1}{a - 1} \mod 2^{128} \tag{3.5}$$

Note that $\nabla S_0 = 0$ and $\nabla S_1 = \Delta S_0$. Therefore, knowledge of ΔS_0 entails that of the whole sequence of ∇S_i. The prediction algorithm we propose proceeds in three phases:

1. Reconstruct $\Delta S_0[0{:}64 + \ell]$ from $X_0, \dots, X_4$, check consistency with $X_5, \dots, X_{63}$.
2. Reconstruct all rotations r_i from this partial knowledge.
3. Fully reconstruct ΔS_0 from the rotations.
4. Reconstruct S_0 from ΔS_0 and the rotations.

Only the first phase is computationally intensive. The four steps are discussed in the next four subsections.

3.4.1 Partial Difference Reconstruction

In order to access to a part of ΔS_i, we use the same "guess-and-determine" strategy as in section 3.3: we guess the least significant bits of S_0 and some rotations, then check consistency. The difference is that, since c is unknown, we must in addition guess the least significant bits of c to obtain the same "long-term advantage" (c is always odd; this makes one less bit to guess). We must also guess $k + 1$ successive rotations to get information on k successive differences ΔS_i.

Confirming that the guesses are correct is less immediate. When c was known, we could reconstruct the internal state; from there, filtering out the bad guesses was easy. When c is unknown, the same strategy does not work, but a very strong consistency check can still be implemented.

We consider again the sequence of internal states $\mathbf{S} = (S_0, S_1, \dots) = \mathrm{LCG}_{128}(S_0, c)$. We will guess the ℓ least-significant bits of S_0 and of c, therefore let us assume that their value is known and denote it by w_0 and c_0. We define $\mathbf{S'} = \mathrm{LCG}_{128}(S_0 - w_0, c - c_0)$ and $\mathbf{K} = \mathrm{LCG}_{128}(w_0, c_0)$ — again, $\mathbf{K}$ is known and $\mathbf{S'} = \mathbf{S} - \mathbf{K}$. This time, the components of $\mathbf{S'}$ do *not* follow a geometric progression; but we still have that the ℓ least significant bits of each S'_i are zero. Set $\Delta S'_i \stackrel{def}{=} S'_{i+1} - S'_i$; $\Delta S'[\ell{:}64 + \ell]$ follows a geometric progression of common ratio a modulo 2^{64} (again). This time, we have to find $\Delta S'_0[\ell{:}64 + \ell]$.

As in section 3.3, we have access to $T_i \stackrel{def}{=} S_i[58{:}64 + \ell]$. We want to subtract the known part to obtain $T'_i \stackrel{def}{=} (S_i \boxminus K_i)[58{:}64 + \ell]$, which is the truncation of S'_i. This again introduces an unknown vector $\mathbf{B}$ of *borrows*, and in fact we can only compute $\widetilde{\mathbf{T'}} = \mathbf{S}[58{:}64 + \ell] \boxminus \mathbf{K}[58{:}64 + \ell]$, with $\widetilde{\mathbf{T'}} = \mathbf{T'} \boxplus \mathbf{B}$. As explained above, to access a geometric sequence, we would like to obtain $\Delta T'_i \stackrel{def}{=} T'_{i+1} - T'_i$, but we can only compute:

$$\Delta \widetilde{T}'_i \stackrel{def}{=} \widetilde{T}'_{i+1} - \widetilde{T}'_i = (T'_{i+1} \boxminus T'_i) \boxplus (B_{i+1} \boxminus B_i)$$

We are thus still in the context of the problem discussed in section 3.2, but this time the "noise" caused by the carries is given by $B_{i+1} - B_i$. Instead of being between $\{-1, 0, 1\}$ it is between

$\{-2, -1, 0, 1, 2\}$, because of that it could be seen as the outputs of a LCG truncated of $k - t + 2$ bits. When the guesses are correct, then Babai's rounding will reconstruct $\Delta\widetilde{\mathbf{S}}'[\ell{:}64+\ell]$ from $\Delta\widetilde{\mathbf{T}}'$. This in turn yields $\Delta S_0[0{:}64+\ell]$.

Once we have found $\Delta S_0[0{:}64+\ell]$, we can compute $\nabla S_i[0{:}64+\ell]$ for any i because eq. (3.5) holds modulo $2^{64+\ell}$; because we have guessed the first rotation and the ℓ least significant bits of the state, using (3.3) we gain access to $S_0[58{:}64+\ell]$; combined with the "differences" ∇S_i, this reveals $S_i[58{:}64+\ell]$ for any i (and we already had $S_i[0{:}\ell]$). This allows us to compute $Y_i[0{:}\ell] = S_i[0{:}\ell] \oplus S_i[64{:}64+\ell]$ for any i. Given a "fresh" output X_i, and assuming that the guesses are correct, then we should have:

$$S_i[0{:}\ell] \oplus S_i[64{:}64+\ell] = (X_i \lll r_i)[0{:}\ell]. \tag{3.6}$$

In particular, if the guesses were correct, then we should have for any i:

$$S_i[0{:}\ell] \oplus S_i[64{:}64+\ell] \in \big\{(X_i \lll r)[0{:}\ell] \mid 0 \leq r < 64\big\}. \tag{3.7}$$

If none of the 64 possible rotations yields a match, then the guesses made beforehand have to be wrong. As a consequence, bad guesses can be filtered with an arbitrarily low probability of false positives, by trying several indices i.

A few details still need to be fleshed out. To be precise, let us assume that we have guessed the ℓ least-significant bits of S_0 (we denote them by w_0) and the first rotation r_0. Set $Y_0 = X_0 \lll r_0$. We obtain the i-th state by $S_i \equiv \nabla S_i \boxplus S_0$; however, because the "middle" of S_0 is unknown, then an unknown *carry* may cross the 64-th bit during the addition and perturb $S_i[64{:}64+\ell]$. As a result, there is an unknown vector $\mathbf{C}$, whose components are either 0 or 1, such that such that:

$$S_i[64{:}64+\ell] = C_i \boxplus \nabla S_i[64{:}64+\ell] \boxplus (\underbrace{w_0 \oplus Y_0[0{:}\ell]}_{S_0[64:64+\ell]})$$

In algorithm 8, ConsistencyCheck uses eq. (3.7) combined with this observation to discard bad guesses.

The heart of the algorithm is again the reconstruction of a truncated geometric progression knowing the $t = \ell + 6$ upper bits of four consecutive terms. Looking at table 3.1, we see that the best choice consists in guessing 5 consecutive rotations and $\ell = 14$ least-significant bits. Therefore, ReconstructPartialDifference does 2^{57} iterations of the inner loop, and succeeds deterministically.

3.4.2 Predicting all the Rotations

Knowing the values of $\Delta S_0[0{:}64+\ell]$ as well as the ℓ least-significant bits of S_0 and c is sufficient to get rid of the nastier feature of PCG64: armed with this knowledge, we can determine all the subsequent rotations deterministically, at negligible cost, using eq (3.6). For each index i, it suffices to try the 64 possible values of r_i; only one should satisfy eq (3.6). The complete pseudo-code is shown in algorithm 9.

It is unlikely that several possible values of r_i match: each value is "checked" on ℓ bits, so an accidental match happens with probability $2^{\ell-6}$. The total number of lists returned by ReconstructRotations then follows a binomial distribution of parameters $2^{\ell-6}, k$. With $\ell = 14$ and $k = 64$, then only one rotation vector should pass the test for $0 \leq i < 64$ on average.

Algorithm 8 Partial difference reconstruction algorithm (when c is unknown).

1: **procedure** ConsistencyCheck($\Delta S_0, w_0, Y_0, X_5, \ldots, X_k$)
2: $\quad v_0 = w_0 \oplus Y_0[0{:}\ell]$ ▷ $v_0 = S_0[64{:}64+\ell]$
3: $\quad$ **for** $i = 5, \ldots, k$ **do**
4: $\quad\quad u_i \leftarrow \Delta S_0(a^i - 1)(a-1)^{-1} \bmod 2^{64+\ell}$ ▷ $u_i = \nabla S_i[0{:}64+\ell]$
5: $\quad\quad w_i = w_0 \boxplus u_i[0{:}\ell]$ ▷ $w_i = S_i[0{:}\ell]$
6: $\quad\quad v_i = v_0 \boxplus u_i[64{:}64+\ell]$ ▷ $S_i[64{:}64+\ell] \in \{v_i, v_i'\}$
7: $\quad\quad v_i' = v_i \boxplus 1$
8: $\quad\quad \mathcal{C}_i \leftarrow \{w_i \oplus (X_i \lll r_i)[0{:}\ell] \mid 0 \le r_i < 64\}$ ▷ Check eq. (3.7)
9: $\quad\quad$ **if** $\{v_i, v_i'\} \cap \mathcal{C}_i = \emptyset$ **then**
10: $\quad\quad\quad$ **return False** ▷ Bad Guesses
11: $\quad$ **return True** ▷ No inconsistency
12:
13: **procedure** ReconstructPartialDifference($X_0, \ldots, X_k$)
14: $\quad$ // *Statement involving j must be repeated for $j = 0, 1, 2, 3, 4$.*
15: $\quad H \leftarrow$ LLL reduction of $G_{4,64}$
16: $\quad \ell \leftarrow 14$
17: $\quad$ **for** $0 \le w_0 < 2^\ell$ and $0 \le c_0 < 2^{\ell-1}$ **do** ▷ Guess least-significant bits
18: $\quad\quad K_j \leftarrow a^j w_0 + (2c_0+1)(a^j-1)(a-1)^{-1} \bmod 2^{128}$ ▷ Known part
19: $\quad\quad$ **for** $0 \le r_0, r_1, r_2, r_3, r_4 < 64$ **do** ▷ Guess rotations
20: $\quad\quad\quad Y_j \leftarrow X_j \lll r_j$ ▷ Undo rotations
21: $\quad\quad\quad T_j \leftarrow (r_j \oplus Y_j[58{:}64]) + 64 \cdot (K_j \oplus Y_j)\,[0{:}\ell]$ ▷ Truncated LCG
22: $\quad\quad\quad \widetilde{T}_j' \leftarrow T_j \boxminus K_i[58{:}64+\ell]$ ▷ Cancel known part
23: $\quad\quad\quad \Delta\widetilde{T}_j' = \widetilde{T}_{j+1}' \boxminus \widetilde{T}_j'$ ▷ Difference (truncated geom. seq.)
24: $\quad\quad\quad (\Delta U_0, \ldots, \Delta U_3) \leftarrow \left\lfloor (\Delta\widetilde{T}_0', \ldots, \Delta\widetilde{T}_3') \cdot 2^{58-\ell} \cdot \widetilde{H}^{-1} \right\rceil \cdot \widetilde{H}$ ▷ CVP
25: $\quad\quad\quad \Delta S_0[0{:}64+\ell] \leftarrow (K_1 \boxminus K_0)\,[0{:}\ell] + 2^\ell \cdot \Delta U_0[0{:}64]$ ▷ Check
26: $\quad\quad\quad$ **if** ConsistencyCheck($\Delta_0, w_0, Y_0, X_5, \ldots, X_k$) **then**
27: $\quad\quad\quad\quad$ **return** $(w_0, c_0, r_0, \ldots, r_4, \Delta S_0)$.

Algorithm 9 Rotations and full difference reconstruction algorithm

1: **function** ReconstructRotations($\Delta S_0, v_0, i, k$)
2: $\quad$ // *Return a list of potential* `[`$r_i, r_{i+1}, \ldots, r_k$`]`; *assume that* $v_0 = S_0[64{:}64+\ell]$
3: $\quad$ **if** $i > k$ **then**
4: $\quad\quad$ **return** `[]` ▷ End recursion
5: $\quad \mathcal{T} \leftarrow$ ReconstructRotations($\Delta S_0, v_0, i+1, k$) ▷ Find all the $(r_{i+1}, \ldots, r_k)$
6: $\quad \mathcal{H} \leftarrow$ `[]` ▷ List of possible r_i's
7: $\quad u_i \leftarrow \Delta S_0(a^i - 1)(a-1)^{-1} \bmod 2^{64+\ell}$ ▷ $u_i = \nabla S_i[0{:}64+\ell]$
8: $\quad w_i = w_0 + u_i[0{:}\ell] \bmod 2^\ell$ ▷ $w_i = S_i[0{:}\ell]$
9: $\quad v_i = v_0 + u_i[64{:}64+\ell] \bmod 2^\ell$ ▷ $S_i[64{:}64+\ell] \in \{v_i, v_i'\}$
10: $\quad v_i' = v_i + 1 \bmod 2^\ell$
11: $\quad$ **for** $0 \le r < 64$ **do** ▷ Try all rotations
12: $\quad\quad$ **if** $w_i \oplus (X_i \lll r)[0{:}\ell] \in \{v_i, v_i'\}$ **then** ▷ Check eq. (3.6)
13: $\quad\quad\quad \mathcal{H} \leftarrow r :: \mathcal{H}$ ▷ New candidate r_i
14: $\quad$ **return** $\{h :: t \mid h \in \mathcal{H}, t \in \mathcal{T}\}$ ▷ Return $\mathcal{H} \times \mathcal{T}$

3.4.3 Full Difference Reconstruction

Using $X_0, X_1, \ldots, X_{63}$, we recover all rotations and thus we recover the 6 most-significant bits of $S_0, S_1, \ldots, S_{63}$. This allows us to compute the 6 most significant bits of the differences ΔS_i between consecutive states (up to missing carries), and we are faced with the problem of reconstructing a 128-bit geometric progression using 63 consecutive outputs truncated to their 6 most-significant bits. There is again an unknown vector of borrows B such that $\Delta S_i[122{:}128] \boxplus C_i = r_{i+1} \boxminus r_i$.

Reconstructing ΔS_0 from the r_i is exactly the problem discussed in section 2.1. This can be done by solving an instance of CVP in dimension 63. We use the off-the-shelf CVP solver embedded in `fplll`: it runs in negligible time.

3.4.4 Complete State Reconstruction

Once all the rotations have be recovered and ΔS_0 has been found entirely, the only thing that remain is to actually find the entire S_0. For this, we use again eq. (3.3), coupled with the “differences”:

$$\begin{aligned} S_i &= S_0 \boxplus \nabla S_i \\ Y_i &= S_i[0{:}64] \oplus S_i[64{:}128]. \end{aligned}$$

The Y_i and ∇S_i are known, $\nabla S_0 = 0$, and the problem consists in recovering S_0. We could probably encode it as an instance of SAT, feed it to a SAT-solver and be done with it.

Nevertheless, here is a detailed recovery procedure which obtain all bits of S_0, from right to left, by exploiting the non-linearity of modular addition. It takes negligible time. Let $\mathcal{C}_i$ the vector of (incoming) carries generated during the addition of S_0 and ∇S_i:

$$\begin{aligned} S_i[j] &= S_0[j] \oplus \nabla S_i[j] \oplus \mathcal{C}_i[j] \\ \mathcal{C}_i[j] &= \begin{cases} 0 & \text{if } j = 0 \\ \mathrm{MAJ}(S_0[j-1], \nabla S_i[j-1], \mathcal{C}_i[j-1]) & \text{if } j > 0 \end{cases} \end{aligned}$$

Combining all the above, we have:

$$Y_i[j] = Y_0[j] \oplus \big(\nabla S_i[j] \oplus \nabla S_i[64+j]\big) \oplus \big(\mathcal{C}_i[j] \oplus \mathcal{C}_i[64+j]\big) \tag{3.8}$$

This useful equation enables an induction process.

- When $j = 0$, the 0-th carries are zero, and therefore eq. (3.8) reveals the 64-th carries:

  $$\mathcal{C}_i[64+j] = \big(Y_0[j] \oplus Y_i[j]\big) \oplus \big(\nabla S_i[j] \oplus \nabla S_i[64+j]\big).$$

- Next, suppose that $\mathcal{C}_i[0{:}j]$, $S_0[0{:}j-1]$, $\mathcal{C}_i[64{:}64+j]$ and $S_0[64{:}64+j-1]$ are known, for all i. We can compute $\mathcal{C}_i[j] \oplus \mathcal{C}_i[64+j]$ for any i using eq. (3.8). We then look a a specific index $\mathrm{i} > 0$ such that

  $$\nabla S_\mathrm{i}[j-1] \neq \mathcal{C}_\mathrm{i}[j-1] \quad \text{and} \quad \nabla S_\mathrm{i}[64+j-1] = \mathcal{C}_\mathrm{i}[64+j-1].$$

  The point is that, thanks to the majority function, $\mathcal{C}_\mathrm{i}[j] = S_0[j-1]$ and $\mathcal{C}_\mathrm{i}[64+j] = \nabla S_\mathrm{i}[64+j-1]$. It follows that:

  $$S_0[j-1] = Y_0[j-1] \oplus Y_\mathrm{i}[j-1] \oplus \big(\nabla S_\mathrm{i}[j-1] \oplus \nabla S_\mathrm{i}[64+j-1] \oplus \nabla S_\mathrm{i}[64+j-1]\big)$$

  From there, we also have $S_0[64+j-1] = Y_0[64+j-1] \oplus S_0[j-1]$, and the j-th carry bits can be computed normally.

The whole procedure is shown in algorithm 10. Note that once S_0 has been found, then all subsequent states can be computed with error using $S_i = S_0 \boxplus \nabla S_i$. In particular, computing S_1 gives c by $c \leftarrow S_1 \boxminus aS_0$. This complete the reconstruction procedure for PCG64.

Algorithm 10 Full state reconstruction algorithm

```
 1: function ReconstructState(ΔS_0, r_0, ..., r_k, X_0, ..., X_k)
 2:     for i = 0, 1, ..., k do                                                    ▷ Setup
 3:         ∇S_i ← ΔS_0(a^i − 1)(a − 1)^{−1} mod 2^128
 4:         Y_i ← X_i ⋘ r_i                                                         ▷ Undo rotations
 5:         C_i[0] ← 0                                                              ▷ Bootstrap induction
 6:         C_i[64] ← (Y_i[0] ⊕ Y_i[j]) ⊕ (∇S_i[j] ⊕ ∇S_i[64 + j])
 7:     for j = 1, 2, ..., 64 do                                                   ▷ Induction
 8:         i ← ⊥                                                                   ▷ Find good index
 9:         for k = 1, 2, ..., k do
10:             if ∇S_k[j − 1] ≠ C_k[j − 1] ∧ ∇S_k[64 + j − 1] = C_k[64 + j − 1] then
11:                 i ← k
12:         if i = ⊥ then                                                           ▷ No suitable indice found?
13:             Abort with Failure
14:                                                                                 ▷ Compute next state bit
15:         S_0[j − 1] ← Y_0[j − 1] ⊕ Y_i[j − 1] ⊕ (∇S_i[j − 1] ⊕ ∇S_i[64 + j − 1] ⊕ ∇S_i[64 + j − 1])
16:         S_0[64 + j − 1] ← Y_0[64 + j − 1] ⊕ S_0[j − 1]
17:         for i = 0, 1, ..., k do                                                 ▷ Compute next carries
18:             C_i[j] ← MAJ(S_0[j − 1], ∇S_i[j − 1], C_i[j − 1])
19:             C_i[64 + j] ← MAJ(S_0[64 + j − 1], ∇S_i[64 + j − 1], C_i[64 + j − 1])
20:     return S_0
```

3.5 Implementation and Practical Results

We have implemented the state reconstruction algorithms described above using a mixture of C (for the computationally expensive parts) and Python (for the rest). We used the `fplll` library [61] to solve CVP instances exactly in dimension 63.

In this section, we briefly outline important aspects of our implementations and present practical results. Our codes are available in the supplementary material as well as online at:

`https://github.com/cbouilla/pcg/`

The designer of PCG was kind enough to send us two sets challenge inputs: one with the default (known) increment and one with a random secret increment. She generated random seeds and provided us with the first outputs of the pseudo-random generator. We were able to reconstruct the seed with an extremely high confidence level, because they re-generate the same outputs. We emailed back the seeds and received confirmation that they were indeed correct.

We have therefore successfully taken the challenge of predicting the output of the PCG64 generator.

The analysis of section 3.2 yields parameters that guarantee that the reconstruction procedure *always* succeeds. In most cases, these parameters are pessimistic. We ran a serie of experiments to

$n = 3$ (section 3.3)	
ℓ	Success proba.
16	≈ 0.125
17	≈ 0.25
18	≈ 0.5
19	≈ 1
20	1 (proved)

$n = 4$ (section 3.4.1)	
ℓ	Success proba.
10	≈ 0.12
11	≈ 0.64
12	≈ 0.995
13	≈ 1
14	1 (proved)

Table 3.2: Empirical success probabilities with smaller parameters.

determine more practical choices: using smaller-than-guaranteed values of ℓ (the number of guessed least-significant bits), we measured the success probability of the state reconstruction procedure. The results are shown in table 3.2.

3.5.1 Known Increment

When the increment c is known, algorithm 7 is all it takes to reconstruct the internal state of the generator and predict it (or output the seed). We implemented it in C, using OpenMP to parallelize the outer loop that guesses the least-significant bits of the state. This yields a simple multi-core implementation. We used the `gcc 8.3.0` compiler.

From section 3.2.1, we know that guessing $\ell = 20$ least-significant bits ensures deterministic success. However, we observed empirically that $\ell = 19$ works with probability ≈ 1, and runs twice as fast. $\ell = 18$ and $\ell = 17$ run with probability $\approx 1/2$ and $\approx 1/4$ respectively, therefore are much less useful. In practice, we used $\ell = 19$.

We ran it on a server equipped with two 16-core Intel Xeon Gold 6130 CPU @ 2.10GHz ("*Skylake*") CPUs. The inner loop does 2^{37} iterations and terminates in 42.3s, which makes 23 core minutes.

These processors operate at a different frequency depending on the number of cores used and the type of instructions executed. Our code uses only scalar instructions, so the CPUs runs at the highest frequency tier when executing it. Using a single software thread per physical core (each core presents two hardware execution contexts, commercially called *HyperThreads*) allows the CPU to run at 2.8Ghz, the maximum "Turbo" frequency on all cores. Using one software thread per hardware thread reduces the frequency to $\approx$ 2.6Ghz, but allows to better saturate the execution units of the CPU and yields a nearly 20% speedup overall.

Therefore the algorithm requires $2^{41.67}$ CPU cycles in total; this makes less than 26 cycles per iteration of the inner loop. We used several implementation tricks to reach this level of efficiency:

- We used the `__uint128_t` type provided by most C compilers to do 128-bit arithmetic when computing S_1 from S_0. Apart from that, the algorithm has been designed to do mostly 64-bit arithmetic, for the sake of efficiency.
- Looking at the algorithm, it is clear that U_1 and U_2 are actually not needed, so we just don't compute them.
- $\widetilde{T}_J$ is a function of w, j and r_J (with $j = 0, 1, 2$). therefore, for each new value of w, we precompute once and for all an array indexed by (J, R_J) of the 192 possible values of $\widetilde{T}_J$.

- Pushing the same idea a bit further, we precompute parts of the matrix-vector product inside the rounding: this computes a linear combination of the rows of G_3^{-1}, in which $\widetilde{T}_j$ is the coefficient of the j-th row. So we precompute the 576 possible products $\widetilde{T}_j \cdot G_3^{-1}[j,k]$.

- We enumerate the possible rotations in lexicographic order. This means that $\widetilde{T}_0$ changes in each iteration while $\widetilde{T}_1$ (resp $\widetilde{T}_2$) changes every 64 (resp 4096) iterations. Therefore, in 98% of the iterations, two-thirds of the matrix-vector product inside the rounding are the same as from the previous iteration. Therefore, we fully compute the matrix-vector product only when r_1 changes and only update it when r_0 changes.

- The rounding operation, when done naively by writing `llround(x)`, is actually a bottleneck: it calls a library function that accounted for about 20% of the total running time. We instead used the following technique, which correctly returns $\lfloor x \rceil$ whenever $|x| < 2^{51}$: This hack exploits the IEEE754 representation of double-precision floats: the mantissa lies in bits [0:52] while the sign bit and the exponents take the 12 most significant bits. Adding $2^{52} + 2^{51}$ forces the mantissa to shift to the correct position and inserts an extra 1 bit at position 51. The two shifts clear the extra bit and the exponent, while correctly expanding the sign bit.

3.5.2 Unknown Increment

When the increment c is known, the internal state of PCG64 can be practically reconstructed from $X_0, \dots, X_{63}$ using the algorithms shown in section 3.4. Only algorithm 8 is computationally expensive; we implemented it in C, while we implemented algorithms 9 and 10 in Python.

We have shown that algorithm 8 is correct when $\ell = 14$. The procedure does $2^{29+2\ell}$ iterations of the inner loop, so decreasing ℓ would really be interesting. Looking at table 3.2, we settle for $\ell = 13$ in the worst case; let T denotes the running time when $\ell = 13$.

It seems that the most promising strategy consists in choosing $\ell = 11$; if the reconstruction procedure fails, then we try again with different inputs. The expected running time of this approach number of trials is $T/(16 \times 0.64) \approx T/10.25$. In our implementation, $T = 200,000$ CPU hours, so the expected running time of the reconstruction procedure is about $20,000$ CPU hours. In fact we were lucky: on the challenge input, the first attempt with $\ell = 11$ succeeded, so the whole process took only $12,500$ CPU hours.

It actually ran in 35 wall-clock minutes using 512 cluster nodes, each equipped with two 20-cores Intel Xeon Gold 6248 @ 2.5Ghz ("*Cascade Lake*"). The actual machine is the `jean-zay` computer located at the IDRIS national computation center. Note that on this particular parallel computer, running the algorithm with $\ell = 13$ would take 10 hours using the same amount of resources, so the whole procedure *is* practical, even in the absolute worst case.

The outer loop of algorithm 8 makes $2^{2\ell-1}$ iterations while the inner loop makes 2^{30} iterations. Using a single hardware execution context, we measured that one of the outer loop takes between 41.5s and 44s (apparently not all nodes of the cluster are running at exactly the same speed, potentially because of "turbo boost" adjustments and thermal constraints). Because of this variability, we implemented a master-slave work distribution pattern, in which a master process dispatches iterations of the outer loop to slave processes. This also made checkpointing very easy. We used MPI for inter-process communication.

With $\ell = 11$, the whole process took $2^{56.74}$ CPU cycles, which makes less than 54 cycles per iteration of the inner loop. We used essentially the same implementation tricks discussed above. However, this time we had to additionally implement the ConsistencyCheck procedure, which is

called in the inner loop. We observed that the set of possible candidate values $\mathcal{C}$ only depends on w_0 (the variable of the outer loop). Therefore, before entering the inner loop, we precompute a bit field of size 2^ℓ describing $\mathcal{C}_i$. To simplify the implementation, we flatten them by computing $\mathcal{C} = \cup_i \mathcal{C}_i$. This slightly increase the probability of false positives, but makes our code slightly simpler.

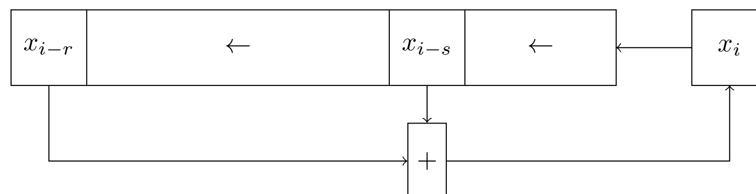
Chapter 4

Attack on Trifork

4.1 Description of Trifork

The generator Trifork has been presented in 2010 by Orue, Montoya, and Hernández Encinas [51] as suitable for cryptographic purposes. The main idea was to construct a fast and secure generator combining three Lagger Fibonacci Generators that are not secure but very fast. To protect itself against attackers, it combines modular arithmetic and binary operations, to avoid arithmetic attacks, and uses very large internal states, to avoid "guess-and-determine". Because of this last characteristic, this generator cannot be used for lightweight cryptography despite its speed and the simplicity of its operations. To keep the size of the key reasonable, this generator has an initialization phase where it uses an LCG to derive the first internal state from three secret words of 64 bits. They claim for their generator a security of 192 bits, hence the size of the key. The proposed algorithm retrieves the seed of this generator in $O(2^{64})$ operations for a large set of parameters. The strategy is to obtain approximations of the outputs of the LCG used in the initialization phase as we already have many tools to attack this generator. This algorithm is an original work firstly presented in *Practical Seed-Recovery of Fast Cryptographic Pseudo-Random Number Generators* at ACNS 2022 [48].

Definition 8. *The Lagged Fibonacci Generator (LFG) is defined by three parameters: r, s and m. The seed contains r words of size $\log_2(m)$: $(x_{-1}, \dots, x_r)$. At step i the generator computes $x_i \equiv x_{i-r} + x_{i-s} \bmod m$. It can be described by the following figure.*



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).

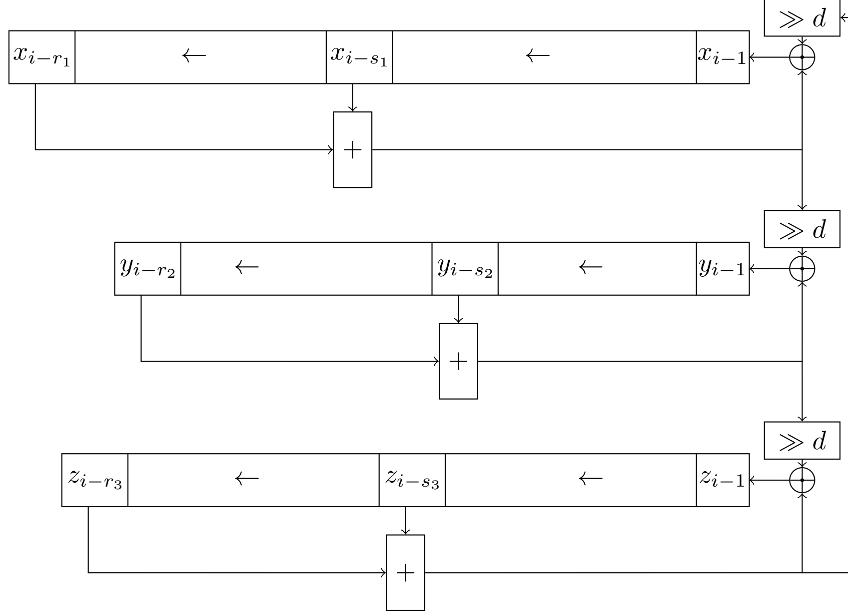


Figure 4.1: Description of Trifork

The Trifork generator, described in Fig. 4.1, is going to use three Lagged Fibonacci Generators of respective parameters $(r_1, s_1, N, 2^n)$, $(r_2, s_2, N, 2^n)$ and $(r_3, s_3, N, 2^n)$. The internal states of the first LFG are denoted (X_i), the internal states of the second one (Y_i), those of the third (Z_i) and the outputs (w_i).

A step i, the generator computes

$$X'_i = X_{i-r_1} + X_{i-s_1} \bmod 2^n$$
$$Y'_i = Y_{i-r_2} + Y_{i-s_2} \bmod 2^n$$
$$Z'_i = Z_{i-r_3} + Z_{i-s_3} \bmod 2^n$$

$$X_i = X'_i \oplus (Z'_i \gg d) \tag{4.1}$$
$$Y_i = Y'_i \oplus (X'_i \gg d) \tag{4.2}$$
$$Z_i = Z'_i \oplus (Y'i \gg d) \tag{4.3}$$

where d is a constant satisfying $0 < d < n$. The output at step i is:

$$W_i = X_i \oplus Z_i.$$

Remark 6. *Trifork uses $r_1 + r_2 + r_3$ words of n bits with $n = 64$. Because it uses Lagged Fibonacci generator, we might want to guess-and-determine the whole internal state (as we will do in chapter 8). The "guess-and-determine" approach consist in guessing some bits of the internal states, using the equations and the known outputs to determine some other bits of the internal state, and then keeping track of these bits to extract some new information the next time their are used to compute*

an output. Here the internal state appears too large to let us hope we could use a classical "guess-and-determine" approach. But because the internal state is that large, it cannot be filled with a secret key (or the secret key would be too large to be usable).

The seed of the generator is $(X_{-r_1}, Y_{-r_2}, Z_{-r_3})$. To fill its internal state, it will use an LCG of public parameters $a, c, 2^n$ with a odd (hence invertible $\mod 2^n$).

For $i \in \{-r_1+1, \dots, -1\}, X_i = aX_{i-1} + c \bmod 2^n$

For $i \in \{-r_2+1, \dots, -1\}, Y_i = aY_{i-1} + c \bmod 2^n$

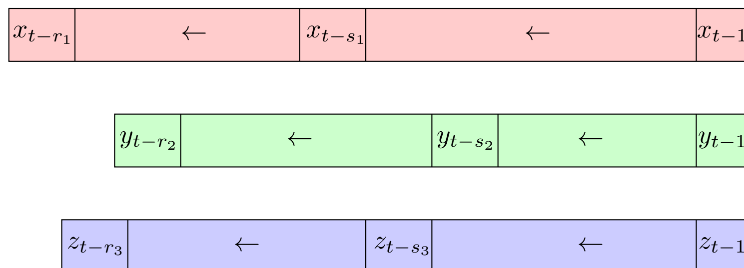
For $i \in \{-r_3+1, \dots, -1\}, Z_i = aZ_{i-1} + c \bmod 2^n$

4.2 General idea behind the attack

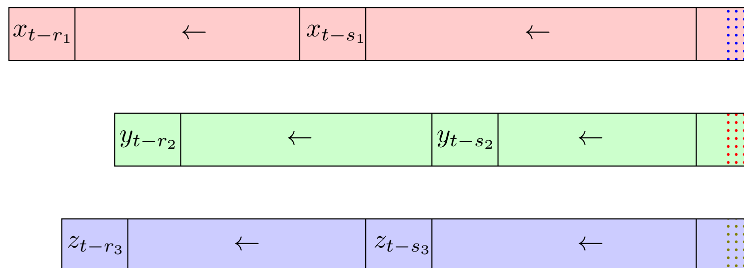
We have seen in chapter 2 that the LCG was not a cryptographic secure PRNG. As it is only used in the initialization phase of Trifork, one could have thought that it was hidden enough not to compromise the security of the whole generator. Alas, the Lagged Fibonacci Generators do not mix the internal states enough to prevent us from attacking this generator.

In the following figure, the words in red depend only on the secret parameter X_{-r_1}, the words in green depend only on the secret parameter Y_{-r_2} and the words in blue depend only on the secret parameter Z_{-r_3}

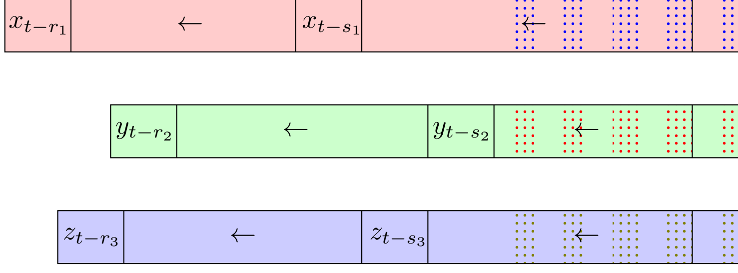
- At $t = 0$, at the end of the initialization, each register depends only on one secret parameter:



- At $t = 1$, only the lower bits of the last words start to depend on two secret parameters at the same time. We see that the words used to compute the next output are still not mixed.



- At $t = s_2 - 1$, the words used to compute the next output are still not mixed (it will be the last time):



By guessing X_{-r_1}, and thus having access to the full first register at $t = 0$, we will manage to reconstruct the upper bits of two sequences that are at least closely related to sequences output from an LCG. Because we need to guess X_{-r_1}, the time complexity of this algorithm will be exponential in n.

4.3 Recovering Z_{-r_3}

We consider a parameter $f_1 \geq s_3$ 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 :

$$g{:}i \to \sum_{j=0}^{i-1} a^j \bmod 2^n \text{ and } f{:}(r,s,i) \to g(r-s+i) + g(i) \bmod 2^n$$

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 $X_i = a^{r_1+i}X_{-r_1} + g(r_1+i) \times c \bmod 2^n$. If $i \geq 0$, by eq (4.1), $\lceil X_i \rceil_d = \lceil X_{i-s_1} + X_{i-r_1} \bmod 2^n \rceil_d$.

- if $i < s_1$, then $\lceil X_i \rceil_d = \lceil a^i(1 + a^{r_1-s_1})X_{-r_1} + f(r_1, s_1, i) \times c \bmod 2^n \rceil_d$ and we can compute this value correctly.
- if $i \geq s_1$, then $\lceil X_i \rceil_d \simeq \lceil X_i - r_1 \rceil_d + \lceil X_{i-s_1} \rceil_d = \lceil a^i X_{-r_1} + g(i) \times c \bmod 2^n \rceil_d + \lceil X_{i-s_1} \rceil_d$ 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 $a^i(1 + a^{r_3-s_3})Z_{-r_3} - t_i = \lfloor Z_i \rfloor_{n-d} \bmod m$. 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

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

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 7. *As we use few outputs, we will not treat the case where $i - r_3 > 0$.*

We set $b = a^{-1} \bmod m$ and $\alpha_3 = (1 + a^{r_3 - s_3})$. We construct

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

which is close to

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

This vector begins like a sequence of outputs of an LCG. We could choose f_1 a bit larger and use only $(t_{s_3}, \ldots, t_{f_1 - 1})$. Then we would only have to attack an LCG as seen earlier in this manuscript. We could also lightly modify the attack using a CVP as seen in subsection 2.1.1.

In this previous attack, we used the fact that we knew the sequence (a_i) satisfying $x_{i+1} \equiv a_i x_i \bmod m$. The additional information about (a_i) following a geometric progression was not used.

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

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

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

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

As seen earlier a CVP solver will retrieve the seed as long as $2\|\mathbf{U} - \mathbf{T}\|_2 < \lambda_1$ and we use the Gaussian heuristic to approach λ_1 by $\sqrt{f_1} 2^{n(f_1 - 1)/f_1}$ (here the seed is $a^{s3} Z_{-r_3}$). As a is invertible we can recover Z_{-r_3}

If $f_1 \leq s_1$, then

$$\begin{aligned}\|\mathbf{U}-\mathbf{T}\|_2 &\leq \sqrt{\sum_{i=0}^{s_3-1}(2^{n-d})^2 + \sum_{i=s_3}^{f_1-1}(2^{n-d+1})^2} \\ &\leq 2^{n-d}\sqrt{s_3 + 4(f_1 - s_3)}\end{aligned}$$

If $s_3 \leq s_1 < f_1$, then

$$\begin{aligned}\|\mathbf{U}-\mathbf{T}\|_2 &\leq \sqrt{\sum_{i=0}^{s_3-1}(2^{n-d})^2 + \sum_{i=s_3}^{s_1-1}(2^{n-d+1})^2 + \sum_{i=s_1}^{f_1-1}(2^{n-d+i-s_1+1})^2} \\ &\leq 2^{n-d}\sqrt{s_1 + 4(s_1 - s_3) + 4\sum_{j=0}^{f_1-s_1-1} 4^j} \\ &\leq 2^{n-d}\sqrt{s_1 + 4(s_1 - s_3) + 4\frac{4^{f_1-s_1}-1}{3}}\end{aligned}$$

If $s_1 < s_3 < f_1$, then

$$\begin{aligned}\|\mathbf{U}-\mathbf{T}\|_2 &\leq \sqrt{\sum_{i=0}^{s_1-1}(2^{n-d})^2 + \sum_{i=s_1}^{s_3-1}(2^{n-d+i-s_1})^2 + \sum_{i=s_3}^{f_1-1}(2^{n-d+i-s_1+1})^2} \\ &\leq 2^{n-d}\sqrt{s_1 + \sum_{j=0}^{s_3-s_1-1} 4^j + 4\sum_{j=s_3-s_1}^{f_1-s_1-1} 4^j} \\ &\leq 2^{n-d}\sqrt{s_1 + \sum_{j=0}^{f_1-s_1-1} 4^j + 3\sum_{j=s_3-s_1}^{f_1-s_1-1} 4^j} \\ &\leq 2^{n-d}\sqrt{s_1 + \frac{4^{f_1-s_1}-1}{4-1} + 3\times 4^{s_3-s_1}\frac{4^{f_1-s_3}-1}{4-1}} \\ &\leq 2^{n-d}\sqrt{s_1 + \frac{4^{f_1-s_1+1}-1}{3} - 4^{s_3-s_1}}\end{aligned}$$

Remark 8. *Alas, the inequality $2\|\mathbf{U}-\mathbf{T}\|_2\sqrt{f_1}2^{n(f_1-1)/f_1}$ is not satisfied in the critical cases. We use an even more loose heuristic and hope $\mathbf{U}$ is indeed the closest vector as long as we have n bits of correct information. If $n/d < s_1$, then we set $f_1 = \max(n/d+1, s3+1$ and the $d-1$ upper bits of the $n/d+1$ computed approximation of X_i are correct. If $n/d \geq s_1$ then we set f_1 such that $f_1 - 1 \times (d - f_1 - s_1) \geq n$. This new heuristic gives similar f_1.*

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

4.4 Recovering Y_{-r_2}

We consider a parameter $f_3 > s_2$ that will be the number of outputs we will use to recover Y_{-r_2}.

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 $\lceil Z_i \rceil_d = \lceil a^i(1 + a^{r_3-s_3})Z_{-r_3} + f(r_3, s_3, i) \times c \bmod 2^n \rceil_d$ 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 bits are computed correctly.

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 $X_i = (a^i(1 + a^{r_1-s_1})X_{-r_1} + f(r_1, s_1, i) \times c \bmod 2^n) \oplus (Z_i \gg d)$.
- if $i \geq s_1$, then $X_i = (a^i X_{-r_1} + g(i) \times c + X_{i-s_1} \bmod 2^n) \oplus (Z_i \gg d)$.

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 $Z_i = (Z_{i-r_3} + Z_{i-s_3} \bmod m) \oplus (Y_i \gg d)$. We call these new values $\bar{Y}_i$.

Remark 9. *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 $t_i = (\bar{Y}_i - \bar{Y}_{i-s_2})2^d - g(i) \times c$.

Remark 10. *Here the dependences between the different values are harder to 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. Thanks 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 $Y_i \gg d$. At this point there is an addition so we might lose one bit of precision because of a carry over. We obtain that $|a^i(1 + a^{r_2-s_2})Y_{-r_2} - t_i| < 2^{n-d+1}$. Because of that we will fix f_3 as follows: 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 $n/d \geq s_3$ then we set f_3 such that $f_3 - 1 \times (d - f_3 - s_3) \geq n$.*

We set $b = a^{-1} \bmod 2^n$ and $\alpha_2 = (1 + a^{r_2-s_2})$. We construct

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

which is close to

$$\mathbf{U} = a^{s2} Y_{-r_2} \times (1, a, a^2, \ldots, a^{f_3-1-s_2}, b^{s_2}\alpha_2 \ldots, b\alpha_2) \bmod 2^n.$$

hence we search for the closest vector to $\mathbf{T}$ in the lattice:

$$\{\beta \times (1, a, a^2, \ldots, a^{f_3-1-s_2}, b^{s_2}\alpha_2 \ldots, b\alpha_2) \bmod 2^n | \beta \in \mathbb{Z}\}.$$

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

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

The CVP solver should return $a^{s_2}Y_{-r_2}$ 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}.

In the end, this attack needs to solve $2^n \times 2$ CVPs on matrices of size $f_1$and f_3 where f_1 and f_3 are small (of size $\simeq n/d$).

4.5 Experimental results

In the original article, a practical instantiation of Trifork was proposed with $n = 64$. As a simple laptop can hardly compute 2^{64} operations, we will present results when X_{-r_1} is known instead of guessed (the computer will only have to compute 2 CVP and a variety of arithmetical operations). The rate of success and time are computed for a hundred of instances, with s_1, s_3, s_3 randomly chosen in $\{1, \dots, 9\}$ and r_1, r_2, r_3 randomly chosen in $\{10, \dots, 20\}$.

d	10	20	30	40	50	60
% of success	22%	99%	100%	100%	59%	0%
time	$0.020s$	$0.018s$	$0.016s$	$0.015s$	$0.015s$	$0.014s$

We notice that our attack does not cover the extreme cases where d is close to one or close to 64. In the case where d is close to 64, the hard problem is to recover the register Y because it will impact the outputs very lightly. But we still should be able to quickly predict a large proportion of output bits without the knowledge of any Y_i. In the case where d is close to one, our method does not allow obtaining precise enough approximations of z_i's.

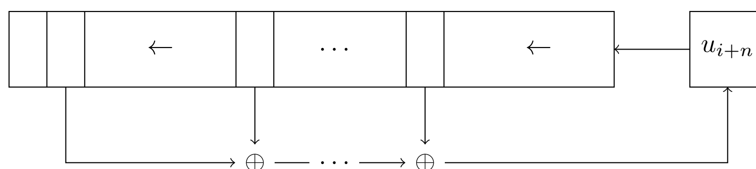
Chapter 5

Attack on the Fast Knapsack Generator

5.1 Description of the Fast Knapsack Generator

In 2009, von zur Gathen and Shparlinski presented a faster and lighter version of the knapsack generator called the *Fast Knapsack Generator* [28]. The main modification was a specialisation of the weights. In their paper, the authors mention that it was not clear if that specialisation had an impact on the security of this generator. Thus it was not known if it was suited for cryptographic purposes. In this chapter, we notice similarities between the fast knapsack generator and the LCG. Because of the specialisation of the weights, the fast knapsack generator tends to act like an LCG on one iteration with probability 1/4. The attack presented here is an original work first presented in *Attacks on Pseudo Random Number Generators Hiding a Linear Structure* at CT-RSA 2022 [47].

Definition 9 (Linear Feedback Shift Register)**.** *The seed of a Linear Feedback Shift Register is made of n bits $u_0, \dots, u_{n-1}$. The public parameter is the feedback polynomial P: an irreducible polynomial over $\mathbb{F}_2$ of degree n. At step i this generator computes $u_{i+n} = P(u_i, \dots, u_{i+n-1})$. It can be represented by the following figure*



Definition 10 (The Knapsack Generator)**.** *An instantiation of the* Knapsack Generator *is given by n secret initial bits $(u_0, \dots, u_{n-1})$, n secret weights $(\omega_0, \dots, \omega_{n-1})$ in $\{0, \dots, 2^{n-1}\}$ and a public feedback polynomial P irreducible over $\mathbb{F}_2$ of degree n. At step i the generator computes:*

$$v_i \equiv \sum_{j=0}^{n-1} u_{i+j}\omega_j \bmod 2^n \text{ and } u_{i+n} = P(u_i, \dots, u_{i+n-1})$$

and the output is made of the $n - \ell$ leading bits of v_i.

We notice that the key is of size $n + n^2$ bits and the generator needs n additions over $\mathbb{Z}/2^n\mathbb{Z}$ to compute a new output. That is why von zur Gathen and Shparlinski introduced the *Fast Knapsack Generator* in 2009 [28] which is lighter and faster than the original Knapsack generator.

Definition 11 (The Fast Knapsack Generator)**.** *An instantiation of the* Fast Knapsack Generator *is given by n secret initial bits $(u_0, \ldots, u_{n-1})$, two secret integers y and z in $\{0, \ldots, 2^{n-1}\}$ and a public feedback polynomial P irreducible over $\mathbb{F}_2$ of degree n. Before producing any outputs the generator computes the weights $\omega_i = z^{n-i}y$ for $i \in \{0, \ldots, n-1\}$. At the first step the generator computes:*

$$v_0 \equiv \sum_{j=0}^{n-1} u_j \omega_j \bmod 2^n \text{ and } u_n = P(u_0, \ldots, u_{n-1}).$$

At step $i + 1$, it computes:

$$v_{i+1} \equiv -u_i z^n y + z v_i + u_{i+n} z y \bmod 2^n \text{ and } u_{i+n} = P(u_i, \ldots, u_{i+n-1})$$

As before the output is given by the $n - \ell$ leading bits of v_{i+1}.

Here the key is of size $3n$, smaller that in first case, and the generator only needs 3 additions over $\mathbb{Z}/2^n\mathbb{Z}$ to compute a new output.

Remark 11. *We obtain this new way to computes v_{i+1} as:*

$$\begin{aligned}
v_{i+1} &\equiv \sum_{j=0}^{n-1} u_{i+j+1}\omega_j \bmod 2^n \\
&\equiv \sum_{j=0}^{n-1} u_{i+(j+1)} z^{n+1-(j+1)} y \bmod 2^n \\
&\equiv z \sum_{k=1}^{n} u_{i+k} z^{n-k} y \bmod 2^n \\
&\equiv z(-u_i z^n y + \sum_{k=0}^{n-1} u_{i+k} z^{n-k} y + u_{i+n} y) \bmod 2^n \\
&\equiv -u_i z^{n+1} y + z v_i + u_{i+n} z y \bmod 2^n
\end{aligned}$$

The control bits (u_i) come from a LFSR. Even if the LFSR is not cryptographically secure, as its characteristic polynomial is irreducible, we can assume that the (u_i) follow a uniform distribution from a statistical viewpoint [50]. Because of that, the case where $v_{i+1} = z v_i \bmod 2^n$ (i.e. $u_i = u_{n+i} = 0$) appears with probability $\frac{1}{4}$.

We are in a case where a PRNG behaves like an LCG with secret multiplier in one iteration with probability 1/4. From section 2.2, we know how to retrieve the multiplier of an LCG with several consecutive outputs and we will present an alternate version of the attack using the coppersmith method of subsection 2.2.1 to retrieve the multiplier of an LCG with several **non consecutive** pairs of consecutive outputs. We will then present two attacks following the same scheme: choosing when we are going to assume the PRNG behaves like an LCG, using an attack against the assumed LCG, obtain a multiplier z and some complete internal states, using the following outputs to guess the y and finally check the consistency.

5.2 Attacking an LCG with non consecutive pairs of outputs

The internal states of the LCG satisfy the following equation

$$x_{i+1} \equiv ax_i + c \bmod N$$

but often we have gaps of unknown size between two outputs. We denote by ℓ the number of discarded bits and by n the bit-size of N. As before $x_i = h_i + \delta_i$ where h_i is construct only with the outputs of the generator and δ_i is unknown and satisfies $|\delta_i| < 2^{\ell-1}$.

Now we suppose we have two pairs of two consecutive internal states (x_0, x_1) and (x_i, x_{i+1}). Then $(\delta_0, \delta_1, \delta_i, \delta_{i+1})$ is a small root of $P \bmod 2^n$ where

$$P(z_0, z_1, z_i, z_{i+1}) = z_0 z_{i+1} - z_1 z_i + h_0 z_{i+1} + h_{i+1} z_0 - h_1 z_i - h_i z_1 + h_0 h_{i+1} - h_1 h_i.$$

We will apply the Coppersmith method on P with $X_0 = X_1 = X_i = X_{i+1} = 2^\ell$. The set of monomials is $\mathfrak{M} = \{z_0, z_1, z_i, z_{i+1}, z_0 z_{i+1}, z_1 z_i\}$ hence, by eq.(1.2), we should heuristically recover the root if $(2^\ell)^8 = X_0 \times X_1 \times X_i \times X_{i+1} \times X_0 X_{i+1} \times X_1 X_i < 2^n$, that is to say if $\ell/n < 1/8$.

Generalisation

Let S be a set of k distinct integers (the larger being i_S) and $\bigcup_{i \in S}\{x_i, x_{i+1}\}$ be at most $2k$ internal states. We will obtain $\binom{k}{2}$ equations of the form $x_j x_{i+1} = x_i x_{j+1} \bmod 2^n$ hence $\binom{k}{2}$ polynomials P_i of which $(\delta_0, \dots, \delta_{i_S+1})$ is a simple root mod 2^n. The set of appearing monomials will be:

$$\mathfrak{M} = \{z_i, z_{i+1} | i \in S\} \bigcup \{z_i z_{j+1} | i, j \in S, i \neq j\}.$$

We will have at most $2k$ monomials of degree 1 and $2\binom{k}{2}$ monomials of degree 2. Heuristically, our attack should work if $(2^\ell)^{2k+4\binom{k}{2}} < (2^n)^{\binom{k}{2}}$. In other words, our attack should work if $\ell/n < \frac{k-1}{4k}$. This theoretical bound increases toward 1/4.

Experimental results

For a given n and k, we search for the greatest ℓ such that the algorithm return the correct multiplier and seed.

k	2	3	4	5	6
$n = 32$					
ℓ (th.)	4	5	6	6	6
ℓ (exp.)	5	8	9	10	11
time	$0.010s$	$0.034s$	$0.098s$	$0.23s$	$0.47s$
$n = 64$					
ℓ (th.)	8	10	12	12	13
ℓ (exp.)	10	16	19	21	22
time	$0.009s$	$0.0.036s$	$0.0.12s$	$0.27s$	$0.57s$
$n = 1024$					
ℓ (th.)	128	170	192	204	213
ℓ (exp.)	170	256	307	341	365
time	$0.012s$	$0.060s$	$0.22s$	$0.76s$	$2.29s$

5.3 Attacking the Fast Knapsack Generator

We consider again the Fast Knapsack Generator. Its internal states satisfy

$$v_{i+1} \equiv -u_i z^{n+1} y + z v_i + u_{i+n} z y \bmod 2^n$$

We construct $H_i = y_i \times 2^\ell + 2^{\ell-1}$ where y_i is the i-th output of the generator, and we denote by δ_i the discarded bits: $\delta_i = v_i - H_i$ and $|\delta_i| < 2^{\ell-1}$. The integer m represent the number of outputs we have.

5.3.1 Attack with consecutive outputs(Coppersmith method)

Finding z: We choose $k+1$ consecutive outputs out of m, hence we choose k steps where we assume the PRNG acts as an LCG. On these $k+1$ outputs H_is we apply the algorithm described in 2.2.1 to attack the underlying LCG and obtain the δ_is completing the $k+1$ chosen outputs (as $v_i = H_i + \delta_i$). If our assumption is false, the δ_is returned by our Coppersmith method might not be integers. If it is the case, we start again with another set of $k+1$ consecutive outputs until the δ_is are integers. Then we can complete our outputs to obtain $k+1$ consecutive internal states. Due to the use of a highly composite modulus 2^n, computing the z is not completely straightforward. If we know v_i and v_{i+1} such that $v_{i+1} = z v_i \bmod 2^n$ we might have to deal with a v_i non-invertible mod 2^n. But usually the exponent of the factor 2 in v_i does not exceed 5 so it is never a problem to do an exhaustive search on the possible values for z.

Finding y: Based on our first assumption, we know z and $k+1$ internal states of the PRNG. We call v_i the last known complete internal state and concentrate on v_{i+1} and v_{i+2}. Based on the structure of the PRNG, there are only 16 possibilities for the relations between v_i, v_{i+1} and v_{i+2}. If these relations are part of the 8 following possibilities, we can recover y again with a Coppersmith method using a lattice of dimension 4.

$$\begin{cases} v_{i+1} &= zv_i + zy \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy \mod 2^n \end{cases} \qquad \begin{cases} v_{i+1} &= zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} &= zv_{i+1} - z^{n+1}y \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} &= zv_i + zy \mod 2^n \\ v_{i+2} &= zv_{i+1} - z^{n+1}y \mod 2^n \end{cases} \qquad \begin{cases} v_{i+1} &= zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} &= zv_i + zy \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy - z^{n+1}y \mod 2^n \end{cases} \qquad \begin{cases} v_{i+1} &= zv_i + zy - z^{n+1}y \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} &= zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy - z^{n+1}y \mod 2^n \end{cases} \qquad \begin{cases} v_{i+1} &= zv_i + zy - z^{n+1}y \mod 2^n \\ v_{i+2} &= zv_{i+1} - z^{n+1}y \mod 2^n \end{cases}$$

For example, let us assume that we are in the first case:

$$\begin{cases} v_{i+1} &= zv_i + zy \mod 2^n \\ v_{i+2} &= zv_{i+1} + zy \mod 2^n. \end{cases}$$

Subtracting the first equation to the second and replacing v_{i+1} by $H_{i+1} + \delta_{i+1}$ and v_{i+2} by $H_{i+2} + \delta_{i+2}$, we obtain:

$$H_{i+2} + \delta_{i+2} - H_{i+1} - \delta_{i+1} = zH_{i+1} + z\delta_{i+1} - zv_i \mod 2^n$$

(we recall that, at this point, v_i and z are assumed to be known). Hence $(\delta_{i+1}, \delta_{i+2})$ is a root of a polynomial in two variables of degree 1 mod2^n. It can be recovered thanks to a Coppersmith method. Once we have v_{i+1}, computing y is straightforward (once again, if the δ_i are not integers it means either our first assumption is false either the couple (v_{i+1}, v_{i+2}) is not of this form).

Remark 12. *There are several little optimisations/improvements we can do in this step. But it is mostly finding more particular cases so, for the sake of simplicity, we decided to not describe them here.*

Checking consistency: We have made a first assumption: the $k+1$ chosen outputs of the PRNG can be seen as truncated outputs of an LCG. We have made a second assumption: (v_{i+1}, v_{i+2}) is of a chosen form between the eight listed possibilities. If y and z are the correct ones, we should be able to check consistency from one to the next (for example H_{i+3} should be given by one of the four following internal states: $zv_{i+2},\ zy + zv_{i+2},\ zv_{i+2} - z^{n+1}y$ or $zy + zv_{i+2} - z^{n+1}y$). If the consistency is not obtained, it means one of our assumptions is false, and we must either change our assumption on (v_{i+1}, v_{i+2}) if we did not explore the eight possibilities, either start again from the beginning with a new set of consecutive outputs.

Analysis of the attack

For a given k, we want to know m the number of outputs needed such that the probability of the PRNG acting as an LCG at least k times in a row is greater than 1/2. To do that we need some probabilities.

Bernoulli trials We suppose that we have n Bernoulli trials, each with a probability of success of p. We want to compute the probability of having a *run* of at least k consecutive successes. We denote this probability $Pr(n,p,k)$.

As we cannot have more successes than trials, if $k > n$ then $Pr(n,p,k) = 0$. If $k = n$, it means all the trials must be successes, hence $Pr(n,p,k) = p^k$.

If $n > k$ we have two excluding possibilities to have k successes. First possibility, a run of k successes happen in the last $n-1$ trials. Second possibility, a run of k successes happen in the k first trial an there is *no* run of k successes in the last $n-1$ trials. It means the first k trials are successes, then the $k+1$-th trial is a failure and there is no run of k successes in the $n-k-1$ remaining trials. Hence the probability of having a run of k successes in n trials when $n > k$ is $Pr(n,p,k) = Pr(n-1,p,k) + p^k \times (1-p) \times (1 - Pr(n-k-1,p,k))$

We fix k and p and consider $S[n] = 1 - Pr(n,p,k)$. We notice that $(S[n])_{n\in\mathbb{N}}$ is a constant-recursive sequence:

$$S[n+1] = S[n] - p^k(1-p)S[n-k-1]$$

of order $k+1$ with initial terms being $S[0] = \cdots = S[k-1] = 1$ and $S[k] = 1 - p^k$.

The explicit values of the sequence are given by $S[n] = C_1(r_1)^n + \cdots + C_{k+1}(r_{k+1})^n$ where the r_i are the roots of the characteristic polynomial $x^{k+1} - x^k + p^k(1-p)$ and the C_i are constants given by the initial terms.

In our case, we have m outputs and we want to know the probability of having $k+1$ consecutive internal states of the form $v_{i+1} = zv_i \bmod 2^n$. Given a v_i, the probability that $v_{i+1} = zv_i \bmod 2^n$ is 1/4. So our problem is to compute the probability of having a run of at least k successes in a sequence of $m-1$ Bernoulli trials, the probability of success of each trial being 1/4.

In the following table we give the minimal values of m such that the probability of having a run of k successes in $m-1$ trials is greater than 1/2.

k	2	3	4	5	6	7	8	10
m	15	58	236	944	3783	15138	60565	969085

(Warning, these values are given by numerical approximations, they might not be exact.)

Once m is greater than the computed bound, we hope there will be a set of $k+1$ consecutive outputs acting like an LCG. The two outputs following the last chosen one need to be in eight possibilities out of sixteen. Again it happens with probability 1/2.

Remark 13. *To compute these probabilities, we assumed we always had two outputs* (v_{i+1}, v_{i+2}) *following our output* v_i*. This is not always the case but this problem can be easily solved by choosing either another known* v_i *or the two preceding values of* v_i *instead of the following ones.*

Hence, for a given k, the attack should work with probability greater than 1/4 if m is greater than what is given in the following table and $l/n < \binom{k}{2}/\Gamma(k)$ (as seen in subsection 2.2.1). In this case we will have to run in the worst case $m-k$ instances of LLL on a lattice of dimension $k+1+3\binom{k}{2}$ and $8(m-k)$ instances of LLL on a lattice of dimension 4, each with entries of size n. The values followed by $(*)$ are estimated values deriving directly from the experimental results of the underlying LCG-seed retriever.

k	2	3	4	5	6
m	15	58	236	944	3783
number of calls to the LCG-solver $\leqslant$	13	55	232	939	3777
$n = 32$					
ℓ (th.) $\leqslant$	4	6	6	6	7
ℓ (exp.) $\leqslant$	6	9	11	12	13$(*)$
time (exp.)	0.19s	1.4s	14s	136s	24 $min(*)$
$n = 64$					
ℓ (th.) $\leqslant$	9	12	13	13	14
ℓ (exp.) $\leqslant$	12	19	22	24	25$(*)$
time (exp.)	0.13s	1.4s	16s	122s	26 $min(*)$
$n = 1024$					
ℓ (th.) $\leqslant$	146	192	211	222	229
ℓ (exp.) $\leqslant$	204	307	361	393	415$(*)$
time (exp.)	0.18s	2.0s	29s	333s	1.8$h(*)$

Exceptionally the experimental time is an average of ten instances of the algorithm.

5.3.2 Attack with consecutive outputs (Stern method)

Finding z: We choose $k+1$ consecutive outputs out of m, hence we choose k steps where we assume the PRNG acts as an LCG. On these $k+1$ outputs H_is we apply the algorithm to attack

the LCG described in subsection 2.2.2 and obtain z. We are going to compute what we assume the internal states are. If we have the right value of z, then the vector of internal states $(v_i, \dots, v_{i+k})$ is in the lattice spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & z & \dots & z^k \\ 0 & 2^n & \dots & 0 \\ & & \ddots & \\ 0 & 0 & 0 & 2^n \end{pmatrix}.$$

Also, this vector is close to the target vector $(H_i, \dots, H_{i+k})$. We use a CVP solver on this matrix and the target vector to find the vector of internal states.

The steps of *Finding y* and *Checking consistency* are the same as for the previous attack.

Analysis of the attack

The number of outputs m is the same as in the previous subsection, as the attack starts in the same way. Hence, for a given k, the attack should work with probability greater than 1/4 if m is greater than what is given in the following table and $l < n(1 - 2/k) + 4 - log_2(k^2 + 4)$ (as seen in subsection 2.2.2). In this case we will have to run in the worst case $m - k$ instances of LLL on a lattice of dimension $k + 1 + 3\binom{k}{2}$ and $8(m - k)$ instances of LLL on a lattice of dimension 4, each with entries of size n.

k	3	4	5	6
m	58	236	944	3783
number of calls to the LCG-solver $\leqslant$	55	232	939	3777
$n = 32$				
ℓ (th.) $<$	5	9	12	13
ℓ (exp.) $\leqslant$	5	11	13	26(∗)
time (exp.)	0.38s	1.7s	8.5s	98s(∗)
$n = 64$				
ℓ (th.) $<$	10	20	25	29
ℓ (exp.) $\leqslant$	10	19	27	32(∗)
time	0.35s	1.6s	9.8s	152s(∗)
$n = 1024$				
ℓ (th.) $<$	170	340	425	
ℓ (exp.) $\leqslant$	168	339	418	512(∗)
time	0.74s	3.3s	15s	162s(∗)

5.3.3 Attack via Coppersmith method without consecutive outputs

Finding z We choose k outputs H_i out of $m - 1$ outputs (we cannot choose the last one) and consider k pairs of outputs (H_i, H_{i+1}). It does not mean we work with $2k$ outputs as some pairs can overlap. On these k pairs of outputs we apply the second algorithm we have against the LCG described in subsection 5.2 and obtain δ_is. If our assumption is false, the δ_is might not be integers. If it is the case, we start again with other sets of k pairs of outputs until the δ_is are integers. Then we can obtain complete internal states (as $v_i = H_i + \delta_i$) to obtain at most $2k$. Computing the z

is not completely straightforward. If we know v_i and v_{i+1} such that $v_{i+1} = zv_i \mod 2^n$ we might have to deal with a v_i non-invertible mod 2^n. But usually the exponent of the factor 2 in v_i does not exceed 5 so it is never a problem to do an exhaustive search on the possible values for z.

The steps of *Finding y* and *Checking consistency* are the same as for the previous attack.

Analysis of the attack

We want the PRNG to act at least k times like an LCG with probability greater than 1/2. We suppose we clock the PRNG $m-1$ times (so we obtain m outputs). The probability that the PRNG acts as an LCG on one iteration is 1/4. Hence we want k to be the unique *median* of a Binomial distribution of parameters $(m-1, 1/4)$. We consider the following theorem from [33].

Theorem 2. *If X is a $B(n,p)$, the median can be found by rounding off np to k if the following condition holds:*

$$|k - np| \leqslant \min(p, 1-p)$$

k is the unique median except when $p = 1/2$ and n is odd.

In the case where $p = 1/4$ we see that given a k the smaller number of trials satisfying this inequality is $4k-1$. Hence, we choose $m = 4k$.

Once m is greater than $4k$, we hope our PRNG will act at least k times like an LCG. The two outputs following the last chosen one need to be in eight possibilities out of sixteen. Again, it happens with probability 1/2.

So for a given k, the attack should work with probability greater than 1/4 if m is greater than $4k$ and $\ell/n < (k-1)/4k$ (as seen in Section 5.2). In this case we will have to run in the worst case $\binom{4k}{k}$ instances of LLL on a lattice of dimension at worst $2k + 3\binom{k}{2}$ and $8\binom{4k}{k}$ instances of LLL on a lattice of dimension 4, each with entries of size n.

Experimental results

k	2	3	4	5	6
m	8	12	16	20	24
number of calls to the LCG-solver $\leq$	21	165	1365	11628	100947
$n = 32$					
ℓ (th.) $<$	4	5	6	6	6
ℓ (exp.) $\leqslant$	5	8	$9(*)$	$10(*)$	$11(*)$
time (exp.)	$0.25s$	$5.2s$	$133s(*)$	$45\ min(*)$	$13h(*)$
$n = 64$					
ℓ (th.) $<$	8	10	12	12	13
ℓ (exp.) $\leqslant$	10	16	$19(*)$	$21(*)$	$22(*)$
time (exp.)	$0.21s$	$5.8s$	$164s(*)$	$52\ min(*)$	$16h(*)$
$n = 1024$					
ℓ (th.) $<$	128	170	192	204	213
ℓ (exp.) $\leqslant$	170	256	$307(*)$	$341(*)$	$365(*)$
time (exp.)	$0.24s$	$7.8s$	$300s(*)$	$2.4h(*)$	$64h(*)$

Here the computing time is an average of ten instances of the algorithm running on the same laptop. As the number of instances of LLL needed is $\binom{4k}{k}$, the computing time of the algorithm quickly explodes.

Chapter 6

Multiple Recursive Generator

The generator we will present in this part is the Multiple Recursive Generator (MRG). It can be seen as a generalization of the LCG. For this reason, we will try to attack this generator using the same attacks we used against the LCG. The goal of the first part was to try to adapt straightforwardly the attacks against the LCG seen in chapter 2. Independently, in [60] and [68] the authors presented a far more detailed and efficient adaptation of the Stern attack presented in section 2.2. Their version even works when the modulus N is unknown. The attack presented at the end of this section on the combined multiple recursive generator is an original attack.

This generator is given by a seed $\mathbf{x}_{seed} = (x_0, \dots, x_{k-1})$ and the equation

$$x_{j+k} = a_{k-1}x_{j+k-1} + \dots + a_0 x_j + c \bmod N$$

where $\mathbf{a} = (a_0, \dots, a_{k-1})$ is the multiplier, c the constant and N the modulus. To obtain the output y_j from an internal state x_j we truncate the ℓ lower bits. We consider $h_j = 2^\ell y_j + 2^{\ell-1}$. We obtain $x_j = h_j + \delta_j$ where $|\delta_j| < 2^{\ell-1}$. As before, we only consider the problem when $c = 0$ as we can get rid of c by considering $x_{j+1} - x_j$ instead.

For redaction purposes, we will give names to plenty of coefficients. As x_j only depends on $\mathbf{x}_{seed}$, we write

$$x_j = b_0^{(j)} x_0 + \dots b_{k-1}^{(j)} x_{k-1} \bmod N$$

where $b_i^{(j)} = 0$ if $j < k$ and $i \neq j$, $b_i^{(j)} = 1$ if $j < k$ and $i = j$ and

$$b_i^{(j)} = \sum_{s=max(0,k-j)}^{k-1} a_s b_i^{(j-k+s)} \bmod N$$

if $j > k$.

6.1 Recovering the seed solving a Closest Vector Problem

We consider $\mathbf{x} = (x_0, \dots, x_m)$ the m first internal states of a PRNG and $\mathbf{h} = (h_0, \dots, h_{m-1})$ its m first outputs. To recover the seed of a LCG, we constructed in subsection 2.1.1 a matrix $L \in \mathcal{M}_{(m \times m)}$ such that $(x_0, n_1, \dots, n_{m-1}) \times L = \mathbf{x}$. To adapt this attack to the MRG it seems

natural to construct a matrix $L \in \mathcal{M}_{(m\times m)}$ such that $(\mathbf{x}_{seed}, n_k, \dots, n_{m-1}) \times L = \mathbf{x}$ where the n_i are integers for the modulus reduction. The rows of the matrix L span the lattice Λ.

$$L = \left(\begin{array}{cccc|cccc} 1 & & & & a_0 & b_0^{(k+1)} & \dots & b_0^{(m-1)} \\ & 1 & & & a_1 & b_1^{(k+1)} & \dots & b_1^{(m-1)} \\ & & \ddots & & \vdots & \vdots & & \vdots \\ & & & 1 & a_{k-1} & b_{k-1}^{(k+1)} & \dots & b_{k-1}^{(m-1)} \\ \hline & & & & N & & & \\ & & & & & N & & \\ & & 0 & & & & \ddots & \\ & & & & & & & N \end{array}\right)$$

The determinant of the lattice is N^{m-k} thus $\lambda_1 \simeq \sqrt{m}N^{(m-k)/m}$. As before if $\mathbf{x}'$ is closer to $\mathbf{h}$ than $\mathbf{x}$ in Λ then:

$$\begin{aligned} \|\mathbf{x} - \mathbf{x}'\|_2 &\leq \|\mathbf{x} - \mathbf{h}\|_2 + \|\mathbf{h} - \mathbf{x}'\|_2 \\ &\leq 2\|\mathbf{x} - \mathbf{h}\|_2 \\ &\leq 2\|\boldsymbol{\delta}\|_2 \\ &\leq 2^{\ell}\sqrt{m} \end{aligned}$$

If we follow the reasoning of subsection 2.1.1, the CVP-solver should return $\mathbf{x}$ from L and $\mathbf{h}$ as long as

$$\ell < n \times (m-k)/m$$

where $n \simeq \log_2(N)$.

6.1.1 Experimental results

For a given n and m we search for the greater ℓ such that the probability of success of retrieving $\mathbf{x}_{seed}$ is above 50%.

For $k = 2$:

m	3	4	5	6	32
$n = 32$					
ℓ (th.) $\leq$	10	16	19	21	30
ℓ (exp.) $\leq$	10	15	18	21	29
time	$0.002s$	$0.002s$	$0.003s$	$0.004s$	$0.12s$
$n = 64$					
ℓ (th.) $\leq$	21	32	38	42	60
ℓ (exp.) $\leq$	21	31	38	42	59
time	$0.001s$	$0.002s$	$0.003s$	$0.004s$	$0.13s$
$n = 1024$					
ℓ (th.) $\leq$	341	512	614	682	960
ℓ (exp.) $\leq$	341	511	614	682	959
time	$0.002s$	$0.003s$	$0.005s$	$0.007s$	$1.58s$

For $k = 3$:

m	4	5	6	7	32
$n = 32$					
ℓ (th.) $\leq$	8	12	16	18	29
ℓ (exp.) $\leq$	8	12	15	18	28
time	$0.002s$	$0.003s$	$0.004s$	$0.006s$	$0.13s$
$n = 64$					
ℓ (th.) $\leq$	16	25	32	36	58
ℓ (exp.) $\leq$	16	25	31	36	57
time	$0.002s$	$0.003s$	$0.005s$	$0.006s$	$0.16s$
$n = 1024$					
ℓ (th.) $\leq$	256	409	512	585	928
ℓ (exp.) $\leq$	256	409	511	584	927
time	$0.003s$	$0.005s$	$0.008s$	$0.01s$	$2.30s$

As before the experimental results seem to confirm our heuristic. This algorithm is fast despite being exponential.

6.2 Retrieving the seed using the attack from Frieze *et al.*

As before, we consider $\mathbf{x} = (x_0, \ldots, x_m)$ the m first internal states of a PRNG and $\mathbf{h} = (h_0, \ldots, h_{m-1})$ its m first outputs. To recover the seed of a LCG using the attack from Frieze *et al.*, we constructed in subsection 2.1.2 a matrix $A \in \mathcal{M}_{(m\times m)}$ such that $A\mathbf{x} \equiv 0 \bmod N$. To adapt this attack to the MRG, it seems natural to construct a matrix $A \in \mathcal{M}_{(m\times m)}$ satisfying the same property.

$$A = \begin{pmatrix} N & & & & & & \\ & \ddots & & & & & \\ & & N & & & & \\ a_0 & \ldots & a_{k-1} & -1 & & & \\ b_0^{(k+1)} & \ldots & b_{k-1}^{(k+1)} & & -1 & & \\ \vdots & & \vdots & & & \ddots & \\ b_0^{(m-1)} & \ldots & b_{k-1}^{(m-1)} & & & & -1 \end{pmatrix}$$

As for the LCG we consider A' the LLL-reduction of A and $\mathbf{c}$ the vector in $\{-N/2, \ldots, N/2\}^m$ satisfying $-A'\mathbf{h} \equiv \mathbf{c} \bmod N$. The new equation is $A'\boldsymbol{\delta} \equiv \mathbf{c} \bmod N$.

If $\|A'\boldsymbol{\delta}\|_\infty < N/2$, the equation is not modular any more and we can compute $\boldsymbol{\delta}$ as $(A')^{-1}\mathbf{c}$.

The determinant of this matrix A' is N^k thus $\|A'\delta\|_\infty \simeq N^{k/m}2^{\ell-1}$. If $N^{k/m}2^{\ell-1} < N/2$ the Frieze algorithm should return $\mathbf{x}_{seed}$. We can simplify the equation as

$$\ell/n \leq \frac{m-k}{m} - \log_2(m)/n$$

where $n \simeq \log_2(N)$.

6.2.1 Experimental results

For a given n and m we search for the greater ℓ such that the probability of success of retrieving the seed $(x_0, \ldots, x_{k-1})$ is above 50%.

For $k = 2$:

m	3	4	5	6	32
$n = 32$					
ℓ (th.) $\leq$	9	14	16	18	25
ℓ (exp.) $\leq$	10	15	18	21	28
time	$0.002s$	$0.002s$	$0.004s$	$0.005s$	$0.12s$
$n = 64$					
ℓ (th.) $\leq$	19	30	36	40	55
ℓ (exp.) $\leq$	21	31	38	42	58
time	$0.002s$	$0.002s$	$0.004s$	$0.006s$	$0.15s$
$n = 1024$					
ℓ (th.) $\leq$	339	510	612	680	955
ℓ (exp.) $\leq$	341	511	614	682	958
time	$0.002s$	$0.003s$	$0.005s$	$0.007s$	$0.35s$

For $k = 3$:

m	4	5	6	7	32
$n = 32$					
ℓ (th.) $\leq$	6	10	13	15	24
ℓ (exp.) $\leq$	8	12	15	18	27
time	$0.003s$	$0.004s$	$0.006s$	$0.007s$	$0.14s$
$n = 64$					
ℓ (th.) $\leq$	14	23	29	33	53
ℓ (exp.) $\leq$	14	25	31	36	56
time	$0.003s$	$0.004s$	$0.006s$	$0.008s$	$0.16s$
$n = 1024$					
ℓ (th.) $\leq$	254	407	509	582	923
ℓ (exp.) $\leq$	255	409	511	584	926
time	$0.003s$	$0.006s$	$0.008s$	$0.01s$	$0.44s$

Once again the results seem to confirm our heuristic (the attainable ℓ is even a bit larger than the heuristic). As expected, the attack is slightly less efficient (we attained smaller ℓ), but faster. In the case ($n = 1024$, $k = 3$ $m = 32$) we go from 2.30s in the CVP attack to 0.44s in the Frieze attack.

6.3 Recovering the seed when the multiplier is unknown

6.3.1 Link with the simplified Stern attack for the Linear Congruential Generator

As before we consider $\mathbf{x} = (x_0, \dots, x_m)$ the m first internal states of a PRNG and $\mathbf{h} = (h_0, \dots, h_{m-1})$ its m first outputs. In the Stern simplified attack on the LCG, in subsection 2.2.2, we applied LLL on the following matrix

$$M_2 = \left(\begin{array}{cccc|ccc} 2^{\ell-1} & & & & & \mathbf{h}_0 & \\ & 2^{\ell-1} & & & & \mathbf{h}_1 & \\ & & \ddots & & & \vdots & \\ & & & 2^{\ell-1} & & \mathbf{h}_{r-1} & \\ \hline & & & & N & & \\ & 0 & & & & \ddots & \\ & & & & & & N \end{array}\right)$$

where $\mathbf{h}_i = (h_i, h_{i+1}, \dots, h_{i+r-1})$.

Let $\boldsymbol{\mu}$ be an integer vector, let $\mathbf{x}_j = (x_j, \dots, x_{j+r})$. We chose parameters such that the only way for $\sum \mu_j \mathbf{h}_j$ to be small was for $\sum \mu_j \mathbf{x}_j \equiv 0 \bmod N$. And if we choose r big enough it would happen only if $\sum \mu_j a^j \equiv 0 \bmod N$

We will try the same heuristic for the MRG

$$\begin{aligned} \sum_{j=0}^{d-1} \mu_j x_{j+s} &\equiv \sum_{j=0}^{d-1} \mu_j \sum_{i=0}^{k-1} b_i^{(j)} x_{i+s} \bmod N \\ &\equiv \sum_{i=0}^{k-1} x_{i+s} \sum_{j=0}^{d-1} \mu_j b_i^{(j)} \bmod N \end{aligned}$$

If $\boldsymbol{\mu}$ satisfies $\forall i \in \{0, \dots, k-1\}$, $\sum_{j=0}^{d-1} \mu_j b_i^{(j)} = 0 \bmod N$, then

$$\mathbf{v} = (2^{l-1}\mu_0, \dots, \mu_{r-1}, \sum \mu_j h_j, \dots, \sum \mu_j h_{j+r-1}) \bmod N$$

is a small vector of M_2 as $\sum \mu_j h_{j+s} \equiv -\sum \mu_j \delta_{j+s} \bmod N$ and the δ_i are small.

Once we have retrieved such a $\boldsymbol{\mu}$, we can construct k polynomials $(P_i)_{i \in \{0,\dots,k-1\}}$ in k variables such that $P_i(a_0, \dots, a_{k-1}) \equiv 0 \bmod N$.

6.3.2 Theoretical parameters

We can expect $\boldsymbol{\mu}$ to be the second part of the smallest vector of the lattice spanned by the matrix $A \in \mathcal{M}_{(d\times d)}$ presented in subsection 6.2. By the Gaussian heuristic, we should have $|\mu_i| \simeq N^{k/r}$. Now we can compute the norm of $\mathbf{v}$ as $\|\mathbf{v}\|_2 \simeq \sqrt{r + r^3/4} \times 2^{\ell-1} N^{k/r}$. Still using the Gaussian

Heuristic, we assume $\lambda_1(M_2) = \sqrt{2r2^{\ell-1}N}$. To have $\|\mathbf{v}\|_2 < \lambda_1(M_2)$ we would need ℓ to satisfy the following equation

$$\ell < n(1 - 2k/r) + 4 - \log_2(4 + r^2)$$

where $n \simeq \log_2(N)$.

6.3.3 Experimental results

For a given n and r we search for the greater ℓ such that the probability of success of retrieving the first multiplier a_0 is above 50%.

- For $k = 2$:

r	5	6	7	8	9
$n = 32$					
ℓ (th.) $\leq$	7	10	13	15	17
ℓ (exp.) $\leq$	6	10	13	15	17
time	$0.15s$	$0.21s$	$0.28s$	$0.37s$	$0.43s$
$n = 64$					
ℓ (th.) $\leq$	13	21	27	31	35
ℓ (exp.) $\leq$	12	20	26	31	34
time	$0.16s$	$0.21s$	$0.29s$	$0.39s$	$0.48s$
$n = 1024$					
ℓ (th.) $\leq$	205	341	438	511	568
ℓ (exp.) $\leq$	204	341	438	511	568
time	$0.25s$	$0.38s$	$0.55s$	$0.75s$	$1.01s$

- For $k = 3$:

r	7	8	9
$n = 32$			
ℓ (th.) $\leq$	4	7	10
ℓ (exp.) $\leq$	3	0	0
time	$0.32s$		
$n = 64$			
ℓ (th.) $\leq$	9	15	20
ℓ (exp.) $\leq$	9	0	0
time	$0.38s$		
$n = 32$			
ℓ (th.) $\leq$	146	255	340
ℓ (exp.) $\leq$	146	0	0
time	$4.8s$		

If we had presented the table giving for which ℓ we can find a polynomial such that $P(a) \equiv 0 \bmod N$ we would have results following quite closely the heuristic. But in the case of the MRG, extracting a and N from those polynomials seems to be hard.

6.4 Multiple Recursive Generator with secret modulus

In the case of the LCG, we were searching for $(\mu_0, \ldots, \mu_{d-1})$ such that for all $j \in \{0, \ldots, r-1\}$, $\sum_{i=0}^{d-1} \mu_i h_{i+j} = 0$. We then expected:

$$\sum_{i=0}^{d-1} \mu_i h_{i+j} \equiv 0 \bmod N \to \sum_{i=0}^{d-1} \mu_i x_{i+j} \equiv 0 \bmod N \to \sum_{j=0}^{d-1} \mu_j a^j \equiv 0 \bmod N$$

In the MRG we do exactly the same thing expecting:

$$\sum_{i=0}^{d-1} \mu_i h_{i+j} \equiv 0 \bmod N \to \sum_{i=0}^{d-1} \mu_i x_{i+j} \equiv 0 \bmod N \to \sum_{j=0}^{d-1} \mu_j b_i^{(j)} \equiv 0 \bmod N$$

but alas the last implication seems to never occur.

6.5 The particular case of Combined Multiple Recursive Generators (CMRG)

These PRNGs output a linear operation between two or more congruential constant-recursive sequences over different moduli, pairwise coprime, of the same length. They have been described in [40]. The coefficients of the sequences and the moduli are known, only the initial conditions are secret. We are going to focus on CMRG outputting the difference between two constant-recursive sequences of order three, $\mathbf{x}$ and $\mathbf{y}$ over two different moduli m_1 and m_2 of the same length n.

At step i, the generator computes

$$\begin{aligned} x_i &= a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3} \mod m_1 \\ y_i &= a_{21}y_{i-1} + a_{22}y_{i-2} + a_{23}y_{i-3} \mod m_2 \\ z_i &= x_i - y_i \mod m_1 \end{aligned}$$

and outputs z_i.

The values a_{11}, a_{12}, a_{13}, , a_{21}, a_{22}, a_{23}, m_1 and m_2 are known. The values x_0, x_1, x_2, y_0,y_1 and y_2 form the seed of the generator.

As m_1 and m_2 are coprime, by the Chinese Reminder Theorem we know that the sequences $\mathbf{x}$ and $\mathbf{y}$ are projections of a lifted constant-recursive sequence modulo m_1m_2 that we will call $\mathbf{X}$. This new sequence will be defined by $X_{i+3} = AX_{i+2} + BX_{i+1} + CX_i \bmod m_1m_2$ where A, B, C are given by:

$$\begin{aligned} A &\equiv a_{11} \bmod m_1 \quad \text{and} \quad A \equiv a_{21} \bmod m_2 \\ B &\equiv a_{12} \bmod m_1 \quad \text{and} \quad B \equiv a_{22} \bmod m_2 \\ C &\equiv a_{13} \bmod m_1 \quad \text{and} \quad C \equiv a_{23} \bmod m_2 \end{aligned}$$

and the initial conditions X_0, X_1, X_2 in $\{0, \ldots, m_1m_2 - 1\}$ satisfy:

$$\begin{aligned} X_0 &\equiv x_0 \bmod m_1 \quad \text{and} \quad X_0 \equiv y_0 \bmod m_2 \\ X_1 &\equiv x_1 \bmod m_1 \quad \text{and} \quad X_1 \equiv y_1 \bmod m_2 \\ X_2 &\equiv x_2 \bmod m_1 \quad \text{and} \quad X_2 \equiv y_2 \bmod m_2. \end{aligned}$$

The sequences $\mathbf{x}$ and $\mathbf{y}$ are given by $\mathbf{x} = \mathbf{X} \bmod m_1$ and $\mathbf{y} = \mathbf{X} \bmod m_2$.

6.5.1 Attack on the MRG32

In 1999, L'Écuyer presented a family of parameters giving CMRGs with good properties [44]. These PRNGs are fast and pass the "spectral test" evaluating their closeness to the uniform distribution. The more famous of these CMRGs is the MRG32k3a, largely used for producing multiple streams of pseudo random numbers, as seen in [45]. It is one of the PRNGs implanted in Matlab and the native PRNG of the programming language Racket. This PRNG had already been used once in place of a secure one for the website *Hacker news*. In 2009, Franke [26] managed to hack this website and was able to steal accounts. His attack was not based on breaking the MRG32k3a but on guessing how the seed was generated. In this case, breaking the MRG32k3a could have led us to another real life attack against this website.

The following attack is an original work first presented in *Attacks on Pseudo Random Number Generators Hiding a Linear Structure* presented at CT-RSA 2022 [47].

Notations: We denote by z'_i the integer value $x_i - y_i$ which can be different from $z_i = x_i - y_i \bmod m_1$. As x_i is already in $\{0, \ldots, m_1 - 1\}$ and y_i is already in $\{0, \ldots, m_2 - 1\}$, we have that $z'_i = z_i$ or $z'_i = z_i - m_1$. We also denote by u the inverse of m_1 modulo m_2 ($um_1 \equiv 1 \bmod m_2$).

Proposition 4. *For every $i \geq 0$, $(x_i, x_{i+1}, x_{i+2}, x_{i+3})$ is a root modulo $m_1 m_2$ of*

$$P_i(v_i, v_{i+1}, v_{i+2}, v_{i+3}) = k_{i+3}m_1 + v_{i+3} - A(k_{i+2}m_1 + v_{i+2}) - B(k_{i+1}m_1 + v_{i+1}) - C(k_i m_1 + v_i)$$

where k_i is the only integer in $\{0, \ldots, m_2 - 1\}$ such that $k_i \equiv -z'_i u \bmod m_2$.

Proof. As $X_i \equiv x_i \bmod m_1$, there exists an integer k_i such that $X_i = k_i m_1 + x_i$. For the same reason, there exists an integer $\hat{k_i}$ such that $X_i = \hat{k_i} m_2 + y_i$. Hence

$$z'_i = x_i - y_i = \hat{k_i} m_2 - k_i m_1.$$

Thus $k_i \equiv -z'_i u \bmod m_2$. As X_i is in $\{0, \ldots, m_1 m_2 - 1\}$, then k_i is in $\{0, \ldots, m_2 - 1\}$. To obtain the polynomial P_i we need to remember that $X_{i+3} = AX_{i+2} + BX_{i+1} + CX_i \bmod m_1 m_2$. □

We have established that (x_0, x_1, x_2, x_3) is a root modulo $m_1 m_2$ of

$$P_1(v_0, v_1, v_2, v_3) = k_3 m_1 + v_3 - A(k_2 m_1 + v_2) - B(k_1 m_1 + v_1) - C(k_0 m_1 + v_0)$$

and each of its coordinates is bounded by m_1.

If this root is the only small one, we can expect to retrieve it thanks to a Coppersmith method. But it tends not to be the case. We will consider Λ the lattice containing all the differences between two roots of P_1 modulo $m_1 m_2$. If the smallest vector v of Λ has its coordinates smaller than m_1, then the vector $(x_0, x_1, x_2, x_3) - v$ could be a smaller root of $P_1 \bmod m_1 m_2$ and our attack might not work.

If we have two roots (x_0, x_1, x_2) and (x'_0, x'_1, x'_2) then

$$(x_3 - x'_3) - A(x_2 - x'_2) - B(x_1 - x'_1) - C(x_0 - x'_0) \equiv 0 \bmod m_1 m_2.$$

Hence the lattice Λ is spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & C \\ 0 & 1 & 0 & B \\ 0 & 0 & 1 & A \\ 0 & 0 & 0 & m_1 m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{4}(m_1 m_2)^{1/4} \approx \sqrt{4} \times 2^{n/2} < \sqrt{4} \times 2^n \approx \sqrt{4} m_1$. Hence, it is unlikely that (x_0, x_1, x_2, x_3) is the only root of P_1 modulo $m_1 m_2$ such that each of its coordinates is bounded by m_1. We try to add other polynomials, hoping it will reduce the number of common roots.

If we consider the three polynomials P_1, P_2 and P_3, the lattice containing the difference between two commons roots will be spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & C & AC & BC + A^2C \\ 0 & 1 & 0 & B & C & B^2 + AC \\ 0 & 0 & 1 & A & (B + A^2) & C + 2AB + A^3 \\ 0 & 0 & 0 & m_1 m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_1 m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_1 m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{6}(m_1 m_2^3)^{1/6} \approx \sqrt{6} \times 2^n \approx \sqrt{6} m_1$. We are at the limit as we have no clear indication that the smallest vector of Λ is big enough. We cannot say that $(x_0, x_1, x_2, x_3, x_4, x_5)$ is the only common root of P_1, P_2 and P_3 modulo $m_1 m_2$ such that each of its coordinates is bounded by m_1. Adding two polynomials was not enough. But the smallest difference between two common roots is far greater than before. So we keep adding polynomials.

If we consider the four polynomials P_1, P_2, P_3 and P_4, the lattice containing the difference between two commons roots will be spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & C & AC & BC + A^2C & C^2 + 2ABC + A^3C \\ 0 & 1 & 0 & B & C & B^2 + AC & 2BC + AB^2 + A^2C \\ 0 & 0 & 1 & A & (B + A^2) & C + 2AB + A^3 & 2AC + B^2 + 2A^2B + A^4 \\ 0 & 0 & 0 & m_1 m_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_1 m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_1 m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & m_1 m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{7}(m_1 m_2^4)^{1/7} \approx \sqrt{7} \times 2^{8n/7} > \sqrt{7} \times 2^n \approx \sqrt{7} m_1$. Hence $(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$ **is** likely to be the only common root of P_1, P_2, P_3 and P_4 modulo $m_1 m_2$ such that each of its coordinates is bounded by m_1. We could wonder if it is relevant to use the Gaussian heuristic in such specific cases, but the parameters given by this reasoning are experimentally recovered.

We can now describe the attack. From a_{11}, a_{12}, a_{13}, , a_{21}, a_{22} and a_{23} we construct A, B and C. Then we consider 7 outputs $z_0, \ldots, z_6$, and from them, we guess $z'_0, \ldots, z'_6$ (we recall that $z'_i = z_i$ or $z'_i = z_i - m_1$). Now we have all the values we need to construct P_1, P_2, P_3 and P_4 as described in Proposition 1.

We use a Coppersmith method to find the only common root of P_1, P_2, P_3 and P_4 mod $m_1 m_2$ with all of its coordinates bound by m_1. If we have correctly guessed the z'_i's, this root has to be $(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$, hence the initial conditions we were searching for. Finally we check the consistency thanks to an eighth output.

Knowing the z_i's we have 2^7 set of possible values for the $z'_i s$. For each set we run one instance of LLL on a lattice of dimension 12 (8 monomials + 4 polynomials) and entries of size n. The time complexity is then $\mathcal{O}(n^3)$.

6.5.2 The MRG32k3a by L'Écuyer

For this particular PRNG, the public values are $m_1 = 2^{32} - 209$, $m_2 = 2^{32} - 22853$, $a_{11} = 0$ $a_{12} = 1403580$, $a_{13} = 810728$, $a_{21} = 527612$, $a_{22} = 0$ and $a_{23} = 1370589$.

If we consider the four polynomials P_1, P_2, P_3, P_4 we find that the smallest difference between two common roots modulo m_1m_2 is (-12600073455, 8717013482, 35458453228, 57149468535, 25239696855, -3505005772, 66309741613). We can see that each of its coordinates is greater than $2 \times m_1$, this ensures that (x_0, x_1, x_2, x_3, x_4, x_5, x_6) will be the only small common root of P_1,P_2,P_3 and P_4 modulo m_1m_2. Our algorithm retrieves the initial conditions in 0.01 second with 8 outputs.

Part III

Attack on a combined generators

Chapter 7

A Generalization of the Knapsack Generator

7.1 Generalized Subset-Sum Generator

In this chapter, we consider a generalization of the subset sum pseudorandom generator, suggested by von zur Gathen and Shparlinski in 2004 [64]. In our abstraction, it is defined by two integer parameters λ and n and three independent components:

- a control-sequence generator CSG:$\{0,1\}^\lambda \times \mathbb{N} \to \{0,1\}^n$;
- an abelian cyclic group $(\mathbb{G}, +)$ of prime order q where the group law is denoted additively;
- a deterministic and public conversion function Ψ:$\mathbb{G} \to \{0,1\}^\rho$ where ρ denotes the output length of the pseudo-random generator.

The seed of this generalized subset-sum generator consists in a bit-string $\mathsf{seed}_0 \in \{0,1\}^\lambda$ and n group elements $P_1, \ldots, P_n \in \mathbb{G}$. The bit size of the seed is thus equal to $\lambda + n \cdot \lceil \log_2(q) \rceil$.

At each iteration $i \in \mathbb{N}$, the control-sequence generator generates an n-bit string $\mathbf{v_i} = (v_i^1, \ldots, v_i^n) = \mathsf{CSG}(\mathsf{seed}_0, i)$, computes the group element Q_i defined by

$$Q_i = [v_i^1]P_1 + \cdots + [v_i^n]P_n \in \mathbb{G}$$

and outputs $s_i = \Psi(Q_i) \in \{0,1\}^\rho$. It is schematized in the figure 7.1 with the secret key in red and the outputs in blue

Example 4. *In the classical knapsack generator, the group $\mathbb{G}$ is the group of modular residue $\mathbb{G} = \mathbb{Z}_m$, the control-sequence generator is defined by a linear feedback shift register and the conversion function Ψ is a truncation.*

In [64], von zur Gathen and Shparlinski proposed to use for $\mathbb{G}$ the group of rational points of an elliptic curve defined over a (prime) finite field, a linear feedback shift register as the control-sequence generator and again a truncation for the conversion function (more precisely, truncation of the abscissa of the elliptic curve point Q_i). They proposed to use $\lambda = n$ and an elliptic curve

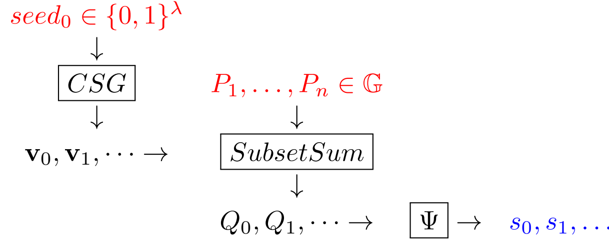


Figure 7.1: Description of the generalized knapsack generator

defined over a finite field $\mathbb{Z}_p$ where p is a n-bit prime number. By the Hasse-Weil theorem, the number of group elements q is around 2^n and the total seed size is $\simeq n + n \cdot n = n \cdot (n+1)$. They suggested that Ψ should discard $\log_2(n)$ low-order bits of the x-coordinate of the point before using it as pseudo-random output and claimed that: "the only available attack on this generator is the brute force search over all parameters defining this generator" and thus using n as small as 12 should provide a 128-bit security level. The statistical properties of the sequences generated by this pseudo-random generator were analyzed in [10, 1, 24].

In this chapter, we present two attacks against this generator (and other variants derived from our abstraction). In the instantiation suggested by von zur Gathen and Shparlinski, our attack has complexity $O(2^{1.778n})$ well below the $O(2^{n(n+1)})$ brute-force attack. We also present a variant in some cases where $\rho = \alpha \cdot n$ with $\alpha < 1$ with a similar complexity.

7.2 High-level description of the attack

We consider the case where the control sequence generated by the CSG is known by the adversary. If this is not the case, they can simply try all possible values for $\mathsf{seed}_0 \in \{0,1\}$ which increases the complexity of the attack by a factor 2^λ.

We assume that the control sequence outputs uniform and independent n-bit strings $\mathbf{v_i} = \mathsf{CSG}(\mathsf{seed}_0, i)$ for each $i \in \mathbb{N}$. Note that this is obviously false but this property does not hold computationally if the control sequence is generated by a shift (as in the classical knapsack generator setting) even if one assumes that the control-sequence generator is a cryptographic pseudo-random generator. We will analyse our attacks using this assumption (and our experimental results will show that it actually holds in practice).

Let us suppose that an adversary finds three indices i_1, i_2, i_3 such that $\mathbf{v_{i_1}} + \mathbf{v_{i_2}} = \mathbf{v_{i_3}}$ as vectors of integers (i.e. where the addition is performed over $\mathbb{Z}$ and not over $\mathbb{Z}_2$). In this case, they know that the relation $Q_{i_1} + Q_{i_2} = Q_{i_3}$ holds in the group $\mathbb{G}$. The adversary is not given the actual values of the points Q_{i_1}, Q_{i_2} and Q_{i_3} but only the values $\Psi(Q_{i_1})$, $\Psi(Q_{i_2})$ and $\Psi(Q_{i_3})$. Assuming that there exist only a few group elements $R_{i_1}^{(1)}, \ldots, R_{i_1}^{(n_1)}$ and $R_{i_2}^{(1)}, \ldots, R_{i_2}^{(n_2)}$ such that $\Psi(R_{i_j}^t) = \Psi(Q_{i_j})$ for $j \in \{1,2\}$ and $t \in \{1, \ldots, n_j\}$ and that the adversary can efficiently retrieve them, they can simply compute $\Psi(R_{i_1}^{t_1} + R_{i_2}^{t_2})$ for all $(t_1, t_2) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\}$ and check whether it is equal to s_{i_3}. If there exists only one such pair (t_1, t_2) then the adversary can safely assumes that $Q_{i_1} = R_{i_1}^{t_1}$, $Q_{i_2} = R_{i_2}^{t_2}$ (and $Q_{i_3} = R_{i_1}^{t_1} + R_{i_2}^{t_2}$).

The number of pairs $(t_1, t_2) \in \{1, \dots, n_1\} \times \{1, \dots, n_2\}$ which satisfy

$$\Psi(R_{i_1}^{t_1} + R_{i_2}^{t_2}) = s_{i_3} \tag{7.1}$$

is difficult to estimate and depends heavily on the group $\mathbb{G}$ and the conversion function Ψ. In [58], Shoup studied the computational complexity of the discrete logarithm in abelian groups in the context of algorithms which do not exploit any special properties of the encodings of group elements. Shoup introduced the *generic group model* where each group element is encoded as a unique and arbitrary binary string (picked uniformly at random and independent of the actual group structure). As a consequence, it is not possible for an algorithm in this model to exploit any special properties of the encodings and group elements can only be operated on using an oracle that provides access to the group operations. If we make a similar assumption on the group $\mathbb{G}$ and if we chose the conversion function Ψ to be a truncation of ℓ bits out of the $(\log_2 q)$-bit encodings of Q_{i_1} and Q_{i_2}, then we can expect the values n_1 and n_2 to be close to 2^ℓ and the number of pairs $(R_{i_1}^{t_1}, R_{i_2}^{t_2})$ different from (Q_{i_1}, Q_{i_2}) satisfying (7.1) to be $2^\ell \cdot 2^\ell / 2^{\log_2(q)-\ell} \simeq 2^{3\ell}/q$. In particular if $\rho > 2 \cdot \log_2(q)/3$, one expects the number of candidates for $(Q_{i_1}, Q_{i_2}, Q_{i_3})$ to be constant in a "generic" group. It is worth mentioning that this assumption does not hold in the classical knapsack generator that uses the group $\mathbb{G} = \mathbb{Z}_m$ since in this case, the number of candidates for a single equation will be about $2^{2\ell}$.

Each relation $\mathbf{v_{i_1}} + \mathbf{v_{i_2}} = \mathbf{v_{i_3}}$ gives two relations in the group $\mathbb{G}$:

$$Q_{i_j} = R_{i_j}^{t_j} = v_{i_j}^1 P_1 + \cdots + v_{i_j}^n P_n$$

for $j \in \{1, 2\}$. If the adversary can recover n points $Q_{i_1}, \dots, Q_{i_n}$ such that $v_{i_1}, \dots, v_{i_n}$ are linearly independent, they would be able to retrieve all the weights used in the generalized knapsack generator.

In the following, we will describe and analyse an algorithm to find "good triplets" of indices (i_1, i_2, i_3) such that $\mathbf{v_{i_1}} + \mathbf{v_{i_2}} = \mathbf{v_{i_3}}$ and show how to use it to attack the elliptic knapsack generator when $\rho = n - \log_2(n)$ (as suggested by von zur Gathen and Shparlinski) but also when $\rho = \alpha \cdot n$ for some $\alpha < 1$ using more extensively the algebraic group law of elliptic curves and the Coppersmith technique.

7.3 Preliminaries

Bounds for Binomial Distributions. Let H denote the binary entropy function, meaning that $H(x) = -x \log_2(x) - (1-x)\log_2(1-x)$, for all $0 < x < 1$. The following standard bounds for the binomial coefficient can be derived from Stirling's formula:

$$\frac{2^{nH(x)}}{\sqrt{8nx(1-x)}} \leq \binom{n}{xn} \leq \frac{2^{nH(x)}}{\sqrt{2\pi nx(1-x)}}, \qquad (0 < x < 1/2) \tag{7.2}$$

Let $X \sim \mathcal{B}(n, p)$ be a binomial random variable. We will use the classical inequality (7.3) given below, a proof of which can be found in [3] amongst others. Here, $D(a, p)$ is the Kullback-Leibler

divergence between an a-coin and a p-coin:

$$\Pr(X \le an) \le \exp(-nD(a,p)) \qquad \text{if } a < p. \tag{7.3}$$
$$\Pr(X \ge an) \le \exp(-nD(a,p)) \qquad \text{if } a > p,$$
$$D(a,p) = a \ln \frac{a}{p} + (1-a) \ln \frac{1-a}{1-p}.$$

If $Y = Y_1 + \cdots + Y_n$ is a sum of binary random variables, we have the "conditional expectation inequality" [54] (see also [38, MPR]):

$$\Pr(Y > 0) \ge \sum_{i=1}^{n} \frac{\mathrm{E}(Y_j)}{\mathrm{E}(Y \mid Y_i = 1)}. \tag{7.4}$$

Elliptic curves. Let p be a prime number (with $p \ge 5$) and let E be an elliptic curve defined over a prime finite field $\mathbb{F}_p$, that is a rational curve given by the following Weierstrass equation

$$E{:}y^2 = x^3 + ax + b$$

for some $a, b \in \mathbb{F}_p$ with $4a^3 + 27b^2 \neq 0$. It is well known that the set $E(\mathbb{F}_p)$ of $\mathbb{F}_p$-rational points (including the special point O at infinity) forms an abelian group with an appropriate composition rule (denoted additively) where O is the neutral element (for more details on elliptic curves, we refer to [11, 65]).

For two points $P = (x_P, y_P) \in E(\mathbb{F}_p)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_p)$, with $P, Q \neq O$, the addition law is defined as $R = (x_R, y_R) = P + Q$ where:

- If $x_P \neq x_Q$, then

$$x_R = m^2 - x_P - x_Q \bmod p, \quad y_R = m(x_P - x_R) - y_P \bmod p \tag{7.5}$$

  where, $m = \frac{y_Q - y_P}{x_Q - x_P} \bmod p$

- If $x_P = x_Q$ but $y_P \neq y_Q$, then $R = O$
- If $P = Q$ and $y_P \neq 0$, then

$$x_R = m^2 - 2x_P \bmod p, \quad y_R = m(x_P - x_R) - y_P \bmod p$$

  where, $m = \frac{3x_Q^2 + a}{2y_P} \bmod p$

- If $P = Q$ and $y_P = 0$, then $R = O$.

For $n \in \mathbb{N}$, $n \ge 2$, we consider *n-th summation polynomial* $f_n = f_n(X_1, X_2, \ldots, X_n)$ introduced by Semaev in [57] such that

$$f_n(x_1, \ldots, x_n) = 0$$

for $x_i \in \overline{\mathbb{F}_p}$ (the algebraic closure of $\mathbb{F}_p$ if and only if there exist $y_1, \ldots, y_n \in \overline{\mathbb{F}_p}$ such that $(x_1, y_1), \ldots, (x_n, y_n) \in E(\overline{\mathbb{F}_p})$ and

$$(x_1, y_1) + \cdots + (x_n, y_n) = O.$$

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
$x+y$	0	0	1	1	1	1	2	2

Table 7.1: Tabulating all solutions of $x + y = z$ for $x, y, z \in \{0, 1\}$

These polynomials have found interesting applications in cryptography (in particular for solving the discrete logarithm problem on elliptic curves defined over finite fields, see [22, 49] and references therein).

The following lemma gives a simple way of calculating them:

Lemma 1. *The n-th Semaev summation polynomial f_n may be defined by:*

$$\begin{aligned}
f_2(X_1, X_2) &= X_1 - X_2 \\
f_3(X_1, X_2, X_3) &= (X_1 - X_2)^2 X_3^2 - 2\left((X_1 + X_2)(X_1 X_2 + a) + 2b\right) X_3 \\
&\quad +(X_1 X_2 - a)^2 - 4b(X_1 + X_2) \\
f_n(X_1, \ldots, X_n) &= Res_X(f_{n-k}(X_1, \ldots, X_{n-k-1}, X), f_{k+2}(X_{n-k}, \ldots, X_n, X)), \\
&\quad n \geq 4 \text{ and } 1 \leq k \leq n-1 \ .
\end{aligned}$$

The polynomial f_n is symmetric and of degree 2^{n-2} in each variable X_i for any $n \geq 3$. The polynomial f_n is absolutely irreducible and we have

$$f_n(X_1, \ldots, X_n) = f_{n-1}^2(X_1, \ldots, X_{n-1}) X_n^{2^{n-2}} + \ldots$$

7.4 Finding "Good Triplets"

Assume that three lists A, B, and C, each of size N, are made of uniformly random n-bit strings. Let Y be the random variable that counts the number of triplets $(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in A \times B \times C$ such that $\mathbf{x} + \mathbf{y} = \mathbf{z}$ when $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ are seen over $\mathbb{Z}^n$ and not modulo 2 . When this relation holds, we call $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ a "good triplet". Our goals in this section are twofold: 1) lower-bound the probability that A, B and C contain a good triplet and 2) design an algorithm to find good triplets efficiently.

As a warm-up, examining the simplest case ($n = 1$) is interesting (cf. Table 7.1). Looking at this table, we see that $\Pr(x + y = z) = 3/8$. We next prove the following

Theorem 1. *We have*

$$\mathrm{E}\,(Y) = N^3 \left(\frac{3}{8}\right)^n ,$$

and

$$\Pr(Y = 0) \leq \frac{1}{N^3} \left(\frac{8}{3}\right)^n + \frac{3}{N} \left(\frac{10}{9}\right)^n + \frac{3}{N^2} \left(\frac{4}{3}\right)^n .$$

Before going into the proof, we discuss the implications. With $N = \alpha(8/3)^{n/3}$, Theorem 1 yields:

$$\Pr(Y = 0) \leq \frac{1}{\alpha^3} + \frac{3}{\alpha}(0.801...)^n + \frac{3}{\alpha^2}(0.69...)^n.$$

u	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$u+v$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$x+y$	0	0	1	1	1	1	2	2	0	0	1	1	1	1	2	2
$u+y$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

u	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$u+v$	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
$x+y$	0	0	1	1	1	1	2	2	0	0	1	1	1	1	2	2
$u+y$	1	1	2	2	1	1	2	2	1	1	2	2	1	1	2	2

Table 7.2: Tabulating $u+v$, $x+y$, $u+y$ for $x, y, z, u, v \in \{0, 1\}$

Therefore, setting $\alpha = 10$ is sufficient to ensure that a good triplet exists with probability 99.9%.

In addition, it follows from Theorem 1 that

$$\Pr(Y = 0) \leq \frac{1}{\mathrm{E}\,(Y)} + \frac{3}{(\mathrm{E}\,(Y))^{1/3}} + \frac{3}{(\mathrm{E}\,(Y))^{2/3}}. \tag{7.6}$$

(this can be obtained by substituting $N = ((8/3)^n \mathrm{E}\,(Y))^{1/3}$ into the inequality and simplifying). This other form is less precise but it is sometimes more practical.

We now proceed to prove the theorem.

Proof. Let x, y, z, u, v denote five independent random bits, and set:

$$\rho = \Pr(x + y = z)$$
$$\sigma = \Pr(u + v = z \mid x + y = z)$$
$$\tau = \Pr(u + y = v \mid x + y = z)$$

We already know that $\rho = 3/8$. Building a simple table as above shows that $\sigma = \tau = 5/12$ (see Table 7.2).

Let $X(i, j, k)$ denote the binary random variable that takes the value 1 if and only if $A[i]+B[j] = C[k]$, so that $Y = \sum X(i, j, k)$. Unless mentioned otherwise, all sums are taken over $0 \leq i, j, k < N$; we omit the indices to alleviate notations.

The expected value of Y is easy to determine. Because the elements of the lists are identically

distributed, $\Pr(A[i]+B[j]=C[k])$ is independent of i, j and k and its value is ρ^n. We get:

$$\begin{aligned} \mathrm{E}\,(Y) = \mathrm{E}\,\left(\sum X(i,j,k)\right) = \sum \mathrm{E}\,(X(i,j,k)) &= \sum \Pr(A[i]+B[j]=C[k]) \\ &= N^3 \left(\frac{3}{8}\right)^n. \end{aligned}$$

Because Y is the sum of binary random variables, we are entitled to use (7.4):

$$\Pr(Y>0) \geq \sum \frac{\mathrm{E}\,(X(i,j,k))}{\mathrm{E}\,(Y \mid X(i,j,k)=1)}.$$

As argued above, the value of the term under the sum is independent of i, j and k, so this boils down to: $\Pr(Y>0) \geq \left(\frac{3}{8}\right)^n / E(Y \mid X(0,0,0)=1)$. It remains to compute the expected number of good triplets under the assumption that there is at least one. This yields:

$$E(Y \mid X(0,0,0)=1) = \sum \Pr(A[i]+B[j]=C[k] \mid A[0]+B[0]=C[0])$$

We split this sum into 8 parts by considering separately the situation where $i=0$, $j=0$ and $k=0$ (resp. $\neq 0$ for each summation index). We introduce the shorthand $p_{ijk} = \Pr\left(A[i]+B[j]=C[k] \mid A[0]+B[0]=C[0]\right)$ and we assume that $i,j,k>0$. Because $A[i]$ is sampled independently from $A[0]$ (resp. B, C), the two events inside the conditional probability are in fact independent and therefore $p_{ijk} = \left(\frac{3}{8}\right)^n$. But when at least one index is zero, this is no longer the case. The extreme situation is $p_{000}=1$.

When there is a single non-zero summation index, the situation is rather simple. If $x+y=z$, then $x+U=z$ if and only if $U=y$, and this happens with probability 2^{-n} because U is uniformly random. This shows that $p_{i00}=p_{0j0}=p_{00k}=2^{-n}$.

It remains to deal with the case of two non-zero summation indices. In fact, p_{ij0} is simply σ^n, while both p_{i0k} and p_{0jk} are equal to τ^n (by the symmetry between the role of the first two lists).

It follows that

$$\begin{aligned} &\mathrm{E}\,(Y \mid X(0,0,0)=1) \\ &= (N-1)^3\left(\frac{3}{8}\right)^n + 3(N-1)^2\left(\frac{5}{12}\right)^n + 3(N-1)\cdot 2^{-n} + 1 \\ &= N^3\left(\frac{3}{8}\right)^n + 3N^2\left(\frac{5}{12}\right)^n + 3N2^{-n} + 1 - \Delta \\ \text{with} \qquad &\Delta = \left(3N^2-3N+1\right)\left(\frac{3}{8}\right)^n + 3(2N-1)\left(\frac{5}{12}\right)^n + 3\cdot 2^{-n}. \end{aligned}$$

The "error term" Δ is always positive for $N \geq 1$. Going back to the beginning, we have:

$$\begin{aligned} \Pr(Y>0) &\geq \frac{N^3(3/8)^n}{N^3(3/8)^n + 3N^2(5/12) + 3N(1/2)^n + 1 - \Delta} \\ &\geq \frac{1}{1+3N^{-1}(10/9)^n + 3N^{-2}(4/3)^n + N^{-3}(8/3)^n} \end{aligned}$$

Using the convexity of $x \longmapsto 1/(1+x)$, we obtain

$$\Pr(Y=0) \leq 3N^{-1}(10/9)^n + 3N^{-2}(4/3)^n + N^{-3}(8/3)^n.$$

$\square$

7.4.1 A Simple Sub-Quadratic Algorithm to Find Good Triplets

Finding a "good triplet" (such that $\mathbf{x}+\mathbf{y}=\mathbf{z}$) can be done using a naive quadratic algorithm: for all pairs $(\mathbf{x},\mathbf{y})$ in $A \times B$, check if $\mathbf{x}+\mathbf{y} \in C$; if so, return it ; after this loop, return $\perp$. This could potentially be sped up a little by exploiting the fact that x and y are necessarily disjoint.

In this section, we present a simple algorithm to find a good triplet more efficiently. We work under the assumption that the input lists have size $N{:}=\alpha(8/3)^{n/3}$ for some constant $\alpha \geq 4$. Under this condition, (7.6) ensures that there is a good triplet with probability at least $\frac{3}{64}$. This assumption will be relaxed in the next section.

Looking again at Fig. 7.1, we see that $\Pr(x=1 \mid x+y=z)=1/3$ while $\Pr(z=1 \mid x+y=z)=2/3$. In other terms, even though x,y,z are sampled uniformly at random, if we restrict our attention to good triplets, then x and y are biased towards zero (sparse) while z is biased towards 1 (dense).

This observation suggests an algorithm to find good triplets efficiently: remove from A, B (resp. C) input vectors of Hamming weight different from $n/3$ (resp. $2n/3$), then run the naive quadratic algorithm on what remains.

Theorem 2. *This algorithm terminates in $\mathcal{O}\left(N^e\right)$ with $e=2\ln(9/4)/\ln(8/3) \approx 1.654$ and succeeds with probability $\Omega\left(\frac{1}{n}\right)$.*

Proof. It follows from the discussion just before the statement of the theorem that there are 3^n good triplets on n bits (out of 8^n triplets in total). The number of good triplets that satisfy the weight condition imposed by the algorithm is

$$N=\binom{n}{n/3,n/3,n/3}=\binom{n}{2n/3}\binom{2n/3}{n/3} \geq \frac{2^{nH(2/3)}}{\sqrt{n}4/3}\frac{2^{2n/3}}{2\sqrt{n/3}}=\frac{3\sqrt{3}}{8n}3^n.$$

If the input list contain a good triplet, then the algorithm described above returns it with probability greater than $0.65/n$. The claimed time complexity is in fact a consequence of the *next* theorem, and we will therefore not prove it here. ☐

7.4.2 Sub-Quadratic Algorithm with Overwhelming Success Probability

We generalize the algorithm of the previous section by relaxing the weight condition. This yields algorithm 11. It takes an additional argument w controlling the maximum allowed weight.

In the sequel, all the stated complexities must be understood "up to a constant factor". Let ϵ denote a constant in the open interval $\left(0;\frac{1}{6}\right)$.

We denote by $\mathrm{wt}(x)$ the Hamming weight of a bit string x.

Algorithm 11 Algorithm to find good triplets.

1: **function** FindTriplet(A,B,C,w)
2: $\quad A' \leftarrow \{x \in A \mid \mathrm{wt}(x) \leq w\}$
3: $\quad B' \leftarrow \{y \in B \mid \mathrm{wt}(y) \leq w\}$
4: $\quad$ **for all** $x,y \in A' \times B'$ **do**
5: $\quad\quad$ **if** $x+y \in C$ **then**
6: $\quad\quad\quad$ **return** (x,y,z)

Lemma 2. *With $w = n\left(\frac{1}{3}+\epsilon\right)$, if the input contains a good triplet, then Algorithm 11 returns $\perp$ with probability less than $2\exp(-2n\epsilon^2)$.*

Proof. Assume that the input lists contain a good triplet (x^*, y^*, z^*). It will be discarded if and only if the weight of either x^*, y^* is greater than w. We know that the weight of x^* and y^* follows a binomial distribution of parameters $(n, 1/3)$, therefore (7.3) shows that either has weight greater than $n(1/3+\epsilon)$ with probability less than $\exp(-nD(1/3+\epsilon, 1/3)$.

The (well-known) fact that $D(p+\epsilon, p) \geq 2\epsilon^2$ combined with union bound (for x^* and y^*) then yields the announced result. □

Lemma 3. *Let T denote the running time of algorithm 11 with $w = n\left(\frac{1}{3}+\epsilon\right)$. Then $\mathrm{E}\,T \leq N + N^2 \exp\left[-nD\left(\frac{1}{3}+\epsilon, \frac{1}{2}\right)\right]$.*

Proof. Filtering the input lists and keeping only low-weight vectors can be done in linear time. Given the complexity of the naive quadratic algorithm, the total time complexity is simply $T = N + |A'| \cdot |B'|$.

Let $X \sim \mathcal{B}(n, 1/2)$ be a binomial random variable modeling the weight of a random n-bit vector. Such a vector belongs to A' or B' if its weight is less than or equal to w, and this happens with probability $s\colon = \Pr(X \leq w)$. The binomial tail bound (7.3) yields the tight upper-bound $s \leq \exp\left[-nD\left(\frac{1}{3}+\epsilon, \frac{1}{2}\right)\right]$.

The sizes of A' and B' are stochastically independent random variables following a binomial distribution of parameters (N, s) with expectation Ns. The expected running time of the quadratic algorithm on A' and B' is therefore $\mathrm{E}\left(|A'| \times |B'|\right) = \mathrm{E}\,|A'| \times \mathrm{E}\,|B'| = N^2 s^2$. Combining this with the upper bound on s gives the announced result. □

Theorem 3. *Write $e = 2 \cdot \dfrac{\ln(9/4)}{\ln(8/3)} \approx 1.654$ For all $d > e$ there is an algorithm that runs in time $\mathcal{O}\left(N^d\right)$, where N denotes the size of the input list and fails to reveal a good triplet present in the input with negligible probability (in n).*

Proof. Let $e < d < 2$ be a complexity exponent greater than the bound e given in the statement of the theorem. There always exist $\epsilon > 0$ such that

$$d = 2 - 6\frac{D\left(\frac{1}{3}+\epsilon, \frac{1}{2}\right)}{\frac{1}{3}\ln\frac{8}{3} + \epsilon \ln 2}.$$

Indeed, setting $\epsilon = 0$ in this expression yields the lower-bound exponent e of the theorem, and the expression of d is increasing as a function of ϵ; it reaches $d = 2$ for $\epsilon = 1/6$.

Let $N_0\colon = (8/3)^{n/3}$, so that input lists of size N_0 contain a single good triplet in average. We distinguish two cases depending of the size of the input lists.

Suppose that $N \leq 2^{\epsilon n} N_0$, where N denotes the size of the input lists. In this case run Algorithm 11 with $w = n\left(\frac{1}{3}+\epsilon\right)$. Lemma 2 guarantees the exponentially small failure probability while lemma 3 tells us that the expected running time T is less than $N + N^2\exp[-2nD\left(\frac{1}{3}+\epsilon, \frac{1}{2}\right)]$.

A quick calculation shows that the algorithm then runs in time $\mathcal{O}\left(N^d\right)$ — the value of d has been chosen for this purpose. The theorem is proved in this case.

If $N > 2^{n\epsilon} N_0$, then slice the input lists in chunks of size $4N_0$ and run Algorithm 11 with $w = n/3$ on each successive chunk until a solution is found. Each chunk contains a good triplet

with probability at least $\frac{3}{64}$ thanks to (7.6). The algorithm reveals this triplet, if it exists, with probability $\Omega\left(\frac{1}{n}\right)$, because it always works if the algorithm of the previous section works.

There are $2^{\epsilon n}/4$ chunks (i.e., exponentially many). Because the chunks are disjoint parts of the input lists, success in a chunk is independent from the others. Therefore the probability that this process fails to reveal a good triplet is negligible. The running time of this procedure is $\mathcal{O}\left(N N_0^{e-1}\right)$. Because $N_0 \leq N$, this is less than $\mathcal{O}\left(N^e\right)$. $\square$

7.5 Practical Key-recovery Attack on von zur Gathen-Shparlinski Elliptic Knapsack Generator

In this section, we consider the instantiation of the knapsack generator suggested by von zur Gathen and Shparlinski in [64]. In particular, the group $\mathbb{G}$ is composed of the points of an elliptic curve E defined over a (prime) finite field $\mathbb{F}_p$ (where $p \geq 5$ is an n-bit prime number). It is a rational curve given by the following Weierstrass equation

$$E\colon y^2 = x^3 + ax + b$$

for some $a, b \in \mathbb{F}_p$ with $4a^3 + 27b^2 \neq 0$. It is well known that the set $E(\mathbb{F}_p)$ of $\mathbb{F}_p$-rational points (including the special point O at infinity) forms an abelian group with an appropriate composition rule (denoted additively) where O is the neutral element — for more details on elliptic curves, we refer to [11, 65]. Von zur Gathen and Shparlinski suggested to use a conversion function $\Psi\colon E \to \{0,1\}^\rho$ that simply truncates $\ell = \log_2(n)$ least significant bits of the abscissa of a point (with $\rho = n - \ell$). An n-bit linear feedback shift register is used as the control-sequence generator (as in the Rueppel-Massy classical knapsack generator) and the overall seed length is thus $n(n+1)$ bits.

7.5.1 Attack on the Elliptic Subset Sum Generator

The adversary first "guesses" seed_0. In other terms, all subsequent steps have to be repeated 2^n times, one for each possible value of seed_0.

Following the analysis from section 7.4, one needs to construct three sets A, B, C of independent vectors $\mathbf{v_i}$ of size $N = 4 \times (8/3)^n$ in order to find a good triplet (i_1, i_2, i_3) such that $\mathbf{v_{i_1}} + \mathbf{v_{i_2}} = \mathbf{v_{i_3}}$ in time $O(N^{1.50019\ldots})$ with probability at least $1 - 1/4^3$. We need to have $n/2$ such good triplets in order to find the n points $P_1, \ldots, P_n$ used as weights in this elliptic knapsack generator, and we can hope to obtain them with constant positive probability from an output sequence made of $O(n^{1/3}N)$ values $s_i \in \{0,1\}^\rho$. Note that in our implementation, we do not distinguish the sets A, B, and C and simply run the algorithm from the previous section with $A = B = C$ the sets of all vectors $\mathbf{v_i}$ corresponding to all known outputs $s_i \in \{0,1\}^\rho$.

Note that as in the classical knapsack generator, the control sequence is not made of independent n-bit strings since if one denotes $(u_n)_{n\geq 0}$ the sequence output by the linear feedback shift register, we have

$$\mathbf{v_i} = (v_i^1, \ldots, v_i^n) = (u_i, u_{i+1}, \ldots, u_{i+n-1}) \in \{0,1\}^n$$

for $i \in \mathbb{N}$. The analysis given in section 7.4 does not apply to such sequences but we make the heuristic assumption that these n-bit tuples are "sufficiently" random and that our algorithm will succeed with a similar probability (this heuristic is shown to be correct by our implementation).

The data complexity of our attack is therefore $O(n^{1/3} \cdot (8/3)^{n/3} \cdot \rho) = O(2^{0.472n})$ bits and finding $n/2$ good triplets with our sub-quadratic algorithm will cost $O((n^{1/3} \cdot N)^e = 2^{2.0.778n})$ operations.

We then follow the general idea given above but for each good triplet (i, j, k) such that $\mathbf{v_i} + \mathbf{v_j} = \mathbf{v_k}$, if the adversary finds two points on the elliptic curve R_i and R_j such that $\Psi(R_i) = s_i$, $\Psi(R_j) = s_j$ and $\Psi(R_i + R_j) = s_k$, then this gives rise to two possible relations:

1. $Q_i = R_i$, $Q_j = R_j$ (and $Q_k = R_i + R_j$), but also
2. $Q_i = -R_i$, $Q_j = -R_j$ (and $Q_k = -(R_i + R_j)$).

This is due to the fact that on an elliptic curve, a point and its negative have representations with much in common since they share the same the x-coordinate (and the y-coordinates are opposites). This "non-genericness" of elliptic curves is well-known and has important consequences in cryptography (e.g. the signature scheme ECDSA is malleable in the sense that if the pair of integers (r, s) is a valid signature of a given message then so is $(r, -s)$). However, with a truncation of $\log_2(n)$ bits of the abscissa of the points, we expect the number of points triple compatible with (s_i, s_j, s_k) to be equal to only 2 (since the algebraic addition law on the elliptic curve is generic compared to the bit-representation of the points except for this negation issue).

Note that for the first such triple, this is not a problem since the generator parametrized with the n points $P_1, \ldots, P_n$ outputs the same sequence as the one parametrized with the n points $-P_1, \ldots, -P_n$. The adversary can then pick up arbitrarily $(Q_i, Q_j) = (R_i, R_j)$ or $(Q_i, Q_j) = (-R_i, -R_j)$. However, for the subsequent relations obtained from other good triplets, the sign choice may be incompatible with the first one and this will result in a system with no solutions. In order to be able to solve the system, we need to have n linear relations among the discrete logarithms of the points $P_1, \ldots, P_n$ and each good triplet gives us two such relations (the third one is by construction a linear combination of the two others and is useless in solving the linear system). Assuming that n is even, one needs to make $n/2 - 1$ choices for the sign of each relation (after the first one), and the adversary can simply make a brute-fore search on all such signs (multiplying the running time of the algorithm by a factor $2^{n/2-1}$).

Once the $n/2$ good triplets have been found, we derive from them n points $Q_{i_1}, Q_{j_1}, \ldots, Q_{i_{n/2}, Q_{j_{n/2}}}$ as seen in previous paragraph and obtain the following linear system:

$$M \times \begin{pmatrix} P_1 \\ \ldots \\ P_n \end{pmatrix} = \begin{pmatrix} Q_{i_1} \\ Q_{j_1} \\ \ldots \\ Q_{i_{n/2}} \\ Q_{j_{n/2}} \end{pmatrix} \text{ with } M = \begin{pmatrix} \mathbf{v}_{i_1} \\ \mathbf{v}_{j_1} \\ \ldots \\ \mathbf{v}_{i_{n/2}} \\ \mathbf{v}_{j_{n/2}} \end{pmatrix}$$

where the unknowns are the P_i's. As the $\mathbf{v}_i$'s are binary vectors, and the matrix M of full rank, it can be easily inverted mod$q$ where q is the order of the elliptic curve E. The secrets weights P_i's are now given by:

$$\begin{pmatrix} P_1 \\ \ldots \\ P_n \end{pmatrix} \equiv M^{-1} \times \begin{pmatrix} Q_{i_1} \\ Q_{j_1} \\ \ldots \\ Q_{i_{n/2}} \\ Q_{j_{n/2}} \end{pmatrix} \bmod q$$

The overall complexity of the attack is thus

$$O\left(\underset{seed_0}{2^n} \times \underset{\text{good triplets}}{(2^{0.778n}} + \underset{\text{derive } Q_i's}{(n/2 \times 2^{2\log_2(n)})} + \underset{\text{inverse } M}{poly(n)} + \underset{\text{sign of } Q_i's}{poly(n) \times 2^{n/2-1})}\right) = O(2^{1.778n})$$

binary operations.

7.5.2 Experimental Results

We first consider the elliptic curve defined by the equation $y^2 = x^3 + 5x + 5$ over $\mathbb{GF}(p)$ where $p = 2^{16} - 15$. This curve contains q points where $q = 65111$. As the curve order is small we have no problem computing discrete logarithms and it takes 23.3 seconds. We present the attack when the control sequence $(\mathbf{v_i})$ is known and we consider $n = 16$ as suggested by von zur Gathen and Shparlinski. The key size in this setting is equal to 256 bits. We should fix $m = 4n^{1/3}(8/3)^{n/3} \simeq 1885$ as seen in the previous subsection but we can use a smaller m at the beginning. We present in the following table the time necessary to recover the secret weights with probability at least 50% when ℓ bits are missing .

ℓ	1	2	3	4	5	6
m	1000	1000	1000	1000	1000	1885
time	$6.9s$	$5.3s$	$5.6s$	$5.02s$	$5.7s$	$26.7s$

For $\ell = 7$ the algorithm stops working because it does not manage to find unique $\Psi(R_1) = s_{i_1}$, $\Psi(R_2) = s_{i_2}$ and $\Psi(R_1 + R_2) = s_{i_3}$. We saw earlier an heuristic predicting the algorithm could not work if $\ell > \log_2(q)/3$ (see eq(7.2). With the current fixed values it becomes $\ell \leqslant 5$ hence our results are coherent with the heuristic.

There is a way to shortcut the exhaustive search on the sign of the points of the elliptic curve. On the first triplet we choose arbitrarily the points R_1, R_2 satisfying $\Psi(R_1) = s_{i_1}$, $\Psi(R_2) = s_{i_2}$ and $\Psi(R_1 + R_2) = s_{i_3}$ (we have two couples possible, (R_1, R_2) and $(-R_1, -R_2)$) .Then we only treat triplets that have at least one index in common with the points we already have. It makes the analysis far more obscure but it keeps on working in practice and is substantially faster.

ℓ	1	2	3	4	5	6
m	1885	1885	1885	1885	1885	1885
time	$1.96s$	$1.99s$	$2.03s$	$2.1s$	$2.46s$	$5.59s$

Now we consider the elliptic curve defined by the equation $y^2 = x^3 + x + 14$ over $\mathbb{GF}(p)$ where $p = 2^{40} + 15$ but still $n = 16$. With this choice we can focus on recovering the points of the elliptic curves from the outputs without being too bothered with finding the good triplets. This curve contains q points where $q = 1099510687747$.

ℓ	1	2	3	4	5	6	7	8	9
m	1885	1885	1885	1885	1885	1885	1885	1885	1750
time	$2.1s$	$2.1s$	$2.08s$	$2.5s$	$2.6s$	$2.1s$	$3.5s$	$8.3s$	$26.7s$

7.6 Theoretical Key-recovery Attack on the Elliptic Knapsack Generator

The attack in the previous section is made possible by the fact that the number of bits removed by the compression function is only logarithmic. By increasing this number substantially and using a compression function that would return only $\rho = \alpha \cdot n$ bits with $0 < \alpha < 1$, the cost of finding the points R_{i_j} for each good triplets would be exponential in n (instead of only polynomial). In this section, we consider a variant of the parameters where:

- the control-sequence generator is a linear feedback shift register with a λ-bit seed;
- the abelian cyclic group $(\mathbb{G}, +)$ is an elliptic curve of prime order q defined over a (prime) finite field $\mathbb{Z}_p$ (but not necessarily with p and q n-bits integers);
- the public conversion function $\Psi{:}\mathbb{G} \to \{0,1\}^\rho$ where $\rho = \lfloor \alpha \cdot \log_2(q) \rfloor$ is simply the truncation of $\lceil (1-\alpha)\log_2(q) \rceil$ bits of the x-coordinate of an elliptic curve point.

A straightforward adaptation of the attack of the previous section gives an attack with complexity

$$O\;\underbrace{(2^\lambda}_{seed_0}\;\times\underbrace{(2^{0.778n}}_{\text{good triplets}}\;\underbrace{+(n/2 \times 2^{2(1-\alpha)\log_2(q)})}_{\text{derive } Q_i's}\;\underbrace{+poly(n) \times 2^{n/2-1})}_{\text{sign of } Q_i's}$$

In this section, we present a lattice-based (heuristic) attack based on Coppersmith's method to improve the part of the complexity $\mathcal{O}(n/2 \times 2^{2(1-\alpha)\log_2(q)})$ in $\mathcal{O}(\log_2(q)))$ for some parameters $\alpha \in]0,1[$.

Given a good triplet (i_1, i_2, i_3) with $\mathbf{v_{i_1}} + \mathbf{v_{i_2}} = \mathbf{v_{i_3}}$, we denote $s_j = s_{i_j}$ the corresponding output of the generator and

$$Q_j = (x_j, y_j) = [v_{i_j}^1]P_1 + \cdots + [v_{i_j}^n]P_n$$

for $j \in \{1,2,3\}$. By definition, we have $x_j = (2^\ell s_j + \gamma_j)$ where $\gamma_j \in \{0, \ldots, 2^\ell - 1\}$ is some value unknown to the adversary (for $j \in \{1,2,3\}$) and p is a $(k+\ell)$ -bit long prime number (with $k = \lfloor \alpha \cdot \log_2(p) \rfloor$). Since (i_1, i_2, i_3) is a good triplet, we have $Q_1 + Q_2 = Q_3$ on the elliptic curve and thus:

$$(x_1 - x_2)^2 x_3^2 - 2\left((x_1 + x_2)(x_1 x_2 + a) + 2b\right) x_3 + (x_1 x_2 - a)^2 - 4b(x_1 + x_2) = 0$$

using the third summation polynomial. By replacing x_j by $(2^\ell s_j + \gamma_j)$ for $j \in \{1,2,3\}$, one obtains a polynomial equation where the coefficients are known to the adversary and that involves the following monomials:

$$\{1, \gamma_1, \gamma_2, \gamma_3, \gamma_1^2, \gamma_1\gamma_2, \gamma_1\gamma_3, \gamma_2^2, \gamma_2\gamma_3, \gamma_3^2, \gamma_1^2\gamma_2, \gamma_1^2\gamma_3, \gamma_1\gamma_2^2, \gamma_1\gamma_2\gamma_3, \gamma_1\gamma_3^2,$$
$$\gamma_2^2\gamma_3, \gamma_2\gamma_3^2, \gamma_1^2\gamma_2^2, \gamma_1^2\gamma_2\gamma_3, \gamma_1^2\gamma_3^2, \gamma_1\gamma_2^2\gamma_3, \gamma_1\gamma_2\gamma_3^2, \gamma_2^2\gamma_3^2\}$$

The sum of degrees of these monomials is equal to

$$1 \times 3 + 2 \times 6 + 3 \times 7 + 4 \times 6 = 60$$

and if one applies Coppersmith's technique to this polynomial (without using shifts or powers of the polynomial) it will succeed if $|\gamma_j| \leq p^{1/60}$ for $j \in \{1,2,3\}$. For $\alpha \geq 59/60$, we thus obtain a (heuristic) attack with the overall complexity

$$O(2^\lambda \cdot (2^{0.778n} + n/2 \cdot \mathsf{poly}(\log_2(q)) + 2^{n/2})).$$

Remark 1. *Note that this attack is mainly theoretical since the bound on α is very close to* 1.

7.7 Practical Key-recovery Attack on the Subset Product Generator

Following the generalization of the knapsack generator to elliptic curves proposed by von zur Gathen and Shparlinski, it is natural to consider other variants using abelian groups of interest in cryptography. The most natural choice is to use (a subgroup of) the multiplicative group of a finite field $\mathbb{Z}_p$ for some prime number p. This group is certainly not generic since there exist sub-exponential time discrete logarithm algorithms in these groups, but it seems that representation of group elements by the unique member of its class in $\{0, \dots, p-1\}$ is sufficiently "generic" that using truncation of their bit-representation as a conversion function would permit an adversary to mount a lattice-based attack on this generator even if a quarter of the bits of each group elements is discarded when computing the output of the generator.

More precisely, in this section, we consider a multiplicative variant of the subset sum generator where:

- the control-sequence generator is a linear feedback shift register with a λ-bit seed;
- the abelian cyclic group $(\mathbb{G}, \cdot)$ is the multiplicative group of a (prime) finite field $\mathbb{Z}_p$ (note that it is denoted multiplicatively);
- the public conversion function $\Psi: \mathbb{G} \to \{0,1\}^\rho$ where $\rho = \lfloor \alpha \cdot \log_2(p) \rfloor$ is simply the truncation of $\lceil (1-\alpha)\log_2(p) \rceil$ bits of the unique member of its group element class in $\{0, \dots, p-1\}$.

We call this generator the *subset product generator*.

7.7.1 Description of the Attack

In this setting, the seed consists in a bit-string $\mathsf{seed}_0 \in \{0,1\}^\lambda$ and n group elements $g_1, \dots, g_n \in \mathbb{Z}_p^*$. The bit size of the seed is thus equal to $\lambda + n \cdot \lceil \log_2(p) \rceil$. At each iteration $i \in \mathbb{N}$, the control-sequence generator generates an n-bit string $v_i = (v_i^1, \dots, v_i^n) = \mathsf{CSG}(\mathsf{seed}_0, i)$, computes the group element h_i defined by

$$h_i = g_1^{v_i^1} \cdots g_n^{v_i^n} \in \mathbb{Z}_p^*$$

and outputs $s_i = \Psi(h_i) = h_i \operatorname{div} 2^\ell \in \{0,1\}^k$ where p is a $(k+\ell)$ -bit long prime number (with $k = \lfloor \alpha \cdot \log_2(p) \rfloor$).

A straightforward adaptation of the attack of the Section 7.5 gives an attack with complexity $O(2^\lambda \cdot (2^{0.78n} + p^{2(1-\alpha)}))$ for $\alpha \geq 2/3$. Note that the complexity does not involve the $O(2^{n/2})$ term that came from the indecision on the signs in the elliptic curve variant of the knapsack generator. We remark that one can improve the complexity of the attack by replacing the brute-force search on the missing bits with the use of Coppersmith technique to retrieve them.

Description of the attack. For a vector v_i output by the control sequence generator, we have

$$h_i = g_1^{v_i^1} \cdots g_n^{v_i^n} \in \mathbb{Z}_p^*$$

with $h_i = (2^\ell s_i + x_i)$ where $x_i \in \{0, \dots, 2^\ell - 1\}$ is some value unknown to the adversary. Given a good triplet (i, j, k) with $v_i + v_j = v_k$, we have $h_i \cdot h_j = h_k \bmod p$ and thus:

$$(2^\ell s_i + x_i) \cdot (2^\ell s_j + x_j) = (2^\ell s_k + x_k) \bmod p.$$

The unknowns (x_i, x_j, x_k) are thus "small" roots of an equation of the form

$$Ax_i + Bx_j + x_i x_j - x_k + C = 0 \bmod p$$

where $A = 2^\ell s_i$, $B = 2^\ell s_j$ and $C = (2^\ell s_i \cdot 2^\ell s_j - 2^\ell s_k) \bmod p$ are values known by the adversary. One can thus apply Coppersmith's technique to this polynomial and the basic technique (without using shifts or powers of the polynomial) will succeed if $|x_i|, |x_j|, |x_k| \leq p^{1/5}$. A simple trick allows us to improve readily this bound by setting $y = x_i x_j - x_k$ such that $|y| \leq 2^{2\ell}$ and solving the equation

$$g(x_i, x_j, y) = Ax_i + Bx_j + y + C = 0 \bmod p$$

in (x_i, x_j, y) is sufficient to recover (x_i, x_j, x_k). Using the basic Coppersmith's technique (again without using shifts or powers of this polynomial), this attack will succeed (heuristically) in polynomial-time if $|x_i|, |x_j|, |x_k| \leq p^{1/4}$. For $\alpha \geq 3/4$, we thus obtain an attack with the overall complexity

$$\mathcal{O}\left(2^\lambda \cdot (2^{0.78n} + n \cdot \mathsf{poly}(\log_2(p)))\right) = \mathcal{O}\left(2^\lambda \cdot (2^{0.78n})\right).$$

Remark 2. *Note that we can improve the bound on the size of the "small" root by using shifts and powers of the polynomial $g(x_i, x_j, y)$. For instance, if one considers the family of four polynomials*

$$\{g, x_i \cdot g, x_j \cdot g, g^2\}$$

that vanish in (x_i, x_j, y) modulo p with total multiplicity $(1+1+1+2) = 5$ and involve the following set of monomials:

$$\{x_i, x_j, y, x_i^2, x_i x_j, x_i y, x_j^2, x_j y, y^2\}$$

with a sum of degrees equal to $(1+1+2+2+2+3+2+3+4) = 20$, we obtain that the Coppersmith's method succeeds (heuristically) if $|x_i|, |x_j|, |x_k| \leq p^{5/20} = p^{1/4}$ (see [32]). This gives the same bound as above. However, if we reintroduce the variable x_k and replace the monomial $x_i x_j$ by $y + x_k$, the total degree of the set of monomials decreases to 19 and this decreases the bound to $p^{5/19}$. It is possible to decrease a bit further the exponent of p in this bound, at the cost of using a lattice of higher dimension in Coppersmith's technique using the technique of unravelled linearization from [30] (see also [8]).

7.7.2 Experimental Results

Exhaustive search on the truncated bits. We consider first the finite field $K = \mathbb{F}_p$ with $p = 2q + 1$ and $q = 99839$. We choose weights in the cyclic multiplicative group G of order q made by the non-quadratic residues of K minus zero. We present the attack when the control sequence (v_i) is known and we consider $n = 16$ as suggested by von zur Gathen and Shparlinski. The key size in this setting is equal to 256 bits. We present in the following table the number m of outputs needed and the time necessary to recover the secret weights with probability at least 50% when ℓ bits are missing.

ℓ	1	2	3	4	5	6
m	1000	1000	1000	1000	1000	1885
time	$0.51s$	$0.45s$	$0.44s$	$0.47s$	$0.58s$	$2.1s$

When 7 bits are truncated we cannot recover the weights even with 1885 outputs.

Now we consider the finite field $K = \mathbb{F}_p$ with $p = 2q + 1$ and

$$q = 72536599031050480402372360602698911648481683373808860129469667649180998227293$$

a 256-bit number, but still $n = 16$. With this choice we can focus on recovering the points from the outputs without being too bothered with finding the good triplets. We need n points to recover the weights and to obtain a good average of the time each computation is run ten times. The whole attack is therefore quite practical.

ℓ	1	2	3	4	5	6	7	8	9
m	1000	1000	1000	1000	1000	1000	1000	1000	1000
time	$0.46s$	$0.50s$	$0.48s$	$0.43s$	$0.55s$	$0.70s$	$0.87s$	$1.9s$	$6.6s$

Coppersmith method. We consider the attack on the second group with $p = 2q + 1$ and q a 256-bit number. First, we implement the attack with the single polynomial $g = Ax_i + Bx_j + y + C$. As the Coppersmith method is a bit more unpredictable, we present in the following table the number m of outputs needed and the time necessary to recover the weights with probability at least 50% when ℓ bits are missing.

ℓ	2	4	8	16	32	62	63
m	1000	1000	1000	1000	1000	1000	1000
time	$0.71s$	$0.67s$	$0.68s$	$0.61s$	$0.63s$	$0.51s$	$0.55s$

If we follow the heuristic in Coppersmith's method we should be able to retrieve the weights up to $\ell = 64$ and $\ell = 64$ is the first instance where the attack stops working. If we try to consider the family of polynomials $\{g, x_i g, x_j g, yg, g^2\}$ instead the improvement on the upper-bound from $p^{1/4}$ to $p^{5/19}$ would not be significant for 256-bit integers.

Chapter 8

Arrow

The attacks presented in this chapter are an original work firstly presented in *Practical Seed-Recovery of Fast Cryptographic Pseudo-Random Number Generators* at ACNS 2022 [48]. They differ from what we have seen before as no lattice-based technique is involved. As the algorithms are long they are not fully described here, they can be found on my git account `https://github.com/floretteM`

8.1 About Lightweight Cryptographic

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 people started looking for lighter PRNGs. In 2017, NIST (National Institute for Standards and Technology) prepared a new competition to standardize algorithms for lightweight cryptography. In [34], they 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. In August 2018, the *call for algorithm* to be considered for lightweight cryptography was published. Since then, NIST received 57 submissions to be considered for standardization. After the initial review of the submissions, 56 were selected as Round 1 candidates. Of the 56 Round 1 candidates, 32 were selected to advance to Round 2. The competition is still ongoing as the time of writing.

8.2 Presentation of Arrow

they had poor statistical properties, which made them easily distinguishable from the uniform distribution, and they were easily predictable (as we could obtain the full internal state by clocking the generator enough times) hence could not be used as cryptographic PRNG. We recall that they are 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 updates its internal state to $(x_{n-r+1}, \ldots, x_n)$.

The goal of Arrow, presented by Lopez, Encinas, Muñoz, and Vitin in [43] in 2017 was to use two 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 binary operations, to break the linearity of the operations. The sequences generated by Arrow pass successfully 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 [12], by Blanco et al. in 2019. As this generator is tailored for lightweight cryptography, its internal states are relatively small.

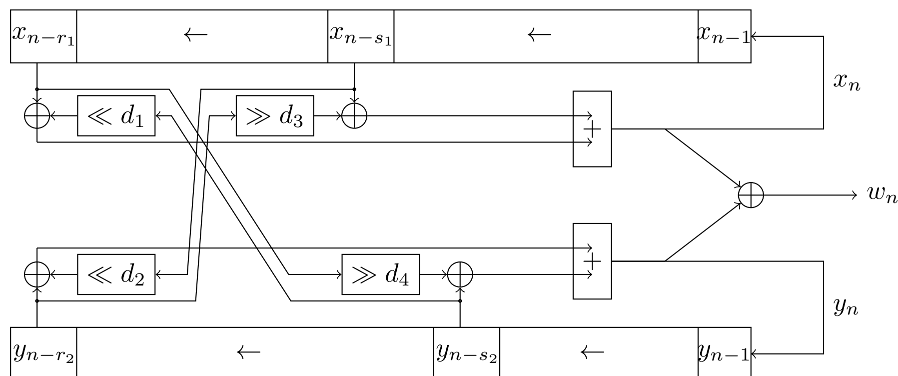


Figure 8.1: Description of Arrow

Arrow is an elaborated architecture, its structure is described in Fig. 8.1. 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 $(x_i)_{-r_1\leq i\leq -1}$ and $(y_i)_{-r_2\leq i\leq -1}$ 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 $n \geq 0$:

$$x_n = ((x_{n-r_1} \oplus (y_{n-s_2} \ll d_1)) + (x_{n-s_1} \oplus (y_{n-r_2} \gg d_3))) \bmod m \tag{8.1}$$

$$y_n = ((y_{n-r_2} \oplus (x_{n-s_1} \ll d_2)) + (y_{n-s_2} \oplus (x_{n-r_1} \gg d_4))) \bmod m \tag{8.2}$$

where d_1, d_2, d_3 and d_4 are four public constant satisfying $0 < d_i < N$. 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 $i \in \{0, \dots, r_2 - 1\}$, 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. The global paradigm were are going to use is call "guess-and-determine". The point it to obtain simple equations on the bits of the internal sates and known parameters/ outputs. Then we *guess* some bits, *determine* other bits using the equations previously obtained and precisely follow how these known bits behave when we clock the internal states and in which future outputs they are going to reappear.

A famous guess-and-determine attack was the real-life attack presented in 1997 against the alleged A5/1, a stream cipher widely used in GSM communications. Several variations of the stream cipher SOBER [53] where also attacked by guess-and-determine attacks such as SOBER-II in 1999 by Bleichenbacher in [13] or SOBER-t32 in 2003 by Baggage *et al.* in [5]. You can find a quick summary of other guess-and-determine attacks in this survey [6], paragraph 3.10.

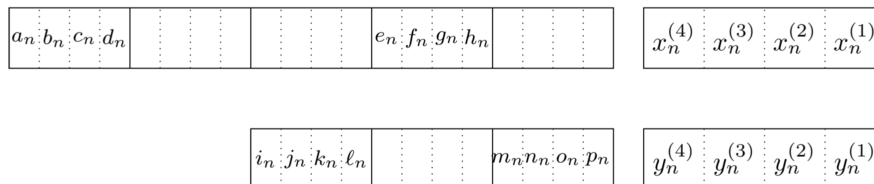
8.3 Attack on a first hardware version of Arrow

We present a first hardware version of Arrow 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	65536	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 (8.1) and (8.2) become:

$$x_n^{(1)} = d_n + (h_n \oplus k_n) \bmod 16 \tag{8.3}$$

$$c_x^{(1)} = (d_n + (h_n \oplus k_n)) \operatorname{div} 16 \tag{8.4}$$

$$x_n^{(2)} = (c_n \oplus p_n) + (g_n \oplus j_n) + c_x^{(1)} \bmod 16 \tag{8.5}$$

$$c_x^{(2)} = ((c_n \oplus p_n) + (g_n \oplus j_n) + c_x^{(1)}) \operatorname{div} 16 \tag{8.6}$$

$$x_n^{(3)} = (b_n \oplus o_n) + (f_n \oplus i_n) + c_x^{(2)} \bmod 16 \tag{8.7}$$

$$c_x^{(3)} = (b_n \oplus o_n) + (f_n \oplus i_n) + c_x^{(2)} \operatorname{div} 16 \tag{8.8}$$

$$x_n^{(4)} = ((a_n \oplus n_n) + e_n + c_x^{(3)}) \bmod 16 \tag{8.9}$$

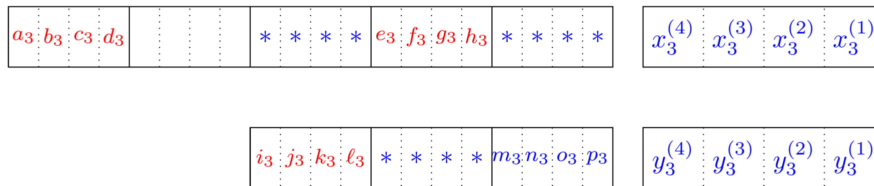
$$y_n^{(1)} = \ell_n + (c_n \oplus p_n) \bmod 16 \tag{8.10}$$
$$c_y^{(1)} = (\ell_n + (c_n \oplus p_n)) \operatorname{div} 16 \tag{8.11}$$
$$y_n^{(2)} = (h_n \oplus k_n) + (b_n \oplus o_n) + c_y^{(1)} \bmod 16 \tag{8.12}$$
$$c_y^{(2)} = ((h_n \oplus k_n) + (b_n \oplus o_n) + c_y^{(1)}) \operatorname{div} 16 \tag{8.13}$$
$$y_n^{(3)} = (g_n \oplus j_n) + (a_n \oplus n_n) + c_y^{(2)} \bmod 16 \tag{8.14}$$
$$c_y^{(3)} = (g_n \oplus j_n) + (a_n \oplus n_n) + c_y^{(2)} \operatorname{div} 16 \tag{8.15}$$
$$y_n^{(4)} = ((f_n \oplus i_n) + m_n + c_y^{(3)}) \bmod 16 \tag{8.16}$$

$$x_n^{(1)} \oplus y_n^{(1)} = w_n^{(1)} \tag{8.17}$$
$$x_n^{(2)} \oplus y_n^{(2)} = w_n^{(2)} \tag{8.18}$$
$$x_n^{(3)} \oplus y_n^{(3)} = w_n^{(3)} \tag{8.19}$$
$$x_n^{(4)} \oplus y_n^{(4)} = w_n^{(4)} \tag{8.20}$$

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. 8.3 and 8.4). Then we compute $y_3^{(1)}$ with $x_3^{(1)}$ and $w_3^{(1)}$ (eq. 8.17) and retrieve p_3 as we know ℓ_3 and c_3 (eq. 8.10). The knowledge of c_3 allows us to compute $x_3^{(2)}$ (eq. 8.5), recover $y_3^{(2)}$(eq. 8.18) and then o_3 (eq. 8.12). With o_3 we can compute $x_3^{(3)}$ (eq. 8.7), recover $y_3^{(3)}$ (eq. 8.19) and then n_3 (eq. 8.14). And finally, with n_3 we can compute $x_3^{(4)}$ (eq. 8.9) and recover $y_3^{(4)}$ (eq. 8.20) as well as m_3 (eq. 8.16). 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 8.17, 8.18, 8.19 and 8.20).

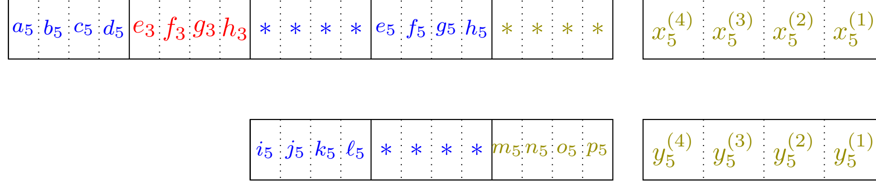


Step 2 We clock the generator twice. As explained above, we have derived a_5, b_5, c_5, d_5 from i_3, j_3, k_3, ℓ_3 and w_0. The values e_5, f_5, g_5, h_5 are $x_3^{(4)}$, $x_3^{(3)}$, $x_3^{(2)}$, $x_3^{(1)}$ and i_5, j_5, k_5, ℓ_5 are m_3,

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. If they are not it means the guesses were wrong and we go back to Step 1 with new guesses. 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 an IV was used or not) 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.

8.4 Another hardware version of Arrow

We study 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 (8.1) and (8.2) become:

a_n b_n					c_n d_n					$x_n^{(2)}$	$x_n^{(1)}$

e_n f_n				g_n h_n				$y_n^{(2)}$	$y_n^{(1)}$

$$x_n^{(1)} = b_n + (e_n \oplus d_n) \bmod 2^{N/2} \tag{8.21}$$

$$c_x = (b_n + (e_n \oplus d_n)) \operatorname{div} 2^{N/2} \tag{8.22}$$

$$y_n^{(1)} = f_n + (a_n \oplus h_n) \bmod 2^{N/2} \tag{8.23}$$

$$c_y = (f_n + (a_n \oplus h_n)) \operatorname{div} 2^{N/2} \tag{8.24}$$

$$x_n^{(2)} = (c_n + (a_n \oplus h_n) + c_x) \bmod 2^{N/2} \tag{8.25}$$

$$y_n^{(2)} = (g_n + (e_n \oplus d_n) + c_y) \bmod 2^{N/2} \tag{8.26}$$

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 0, 12 bits in Step 1, 4 bits in Step 2, and 6 bits in Step 3, 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 has been implemented in C and is running in 20 minutes over 8 threads and with the -O3 option on a standard laptop.

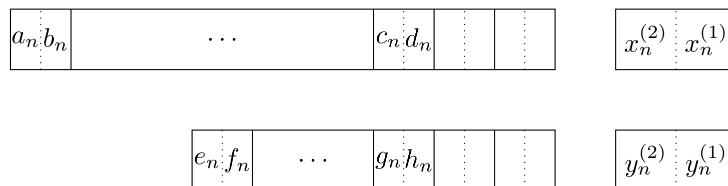
8.5 A software version of Arrow

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 two lagged Fibonacci generator used in this version of Arrows are more or less synchronized (which is something that should have been avoided). If we call t the difference between r_1 and r_2, we notice that $t = r_1 - r_2 = s_2 - r_2$. Hence, if we know e_i, f_i, c_i, d_i we will know $a_{i+14}, b_{i+14}, e_{i+14}, f_{i+14}$. It will ease our guess-and-determine attack;

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): $a_{31} = \bar{e}_{17}, b_{31} = \bar{f}_{17}, e_{31} = g_{17}, f_{31} = h_{17}$
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 $x_{48} \oplus y_{48} = w_{48}$

In step 0, there are $2^{5N/2}$ possibilities for the set of values $\{a_{17}, e_{17}, f_{17}, g_{17}, h_{17}\}$. Thanks to the first filter, on average only $2^{4N/2}$ possibilities are still on course for step 1.

In step 1, there are $2^{6N/2}$ possibilities for the set of values $\{a_{17}, e_{17}, f_{17}, g_{17}, h_{17}, g_{31}, h_{31}\}$ ($2^{4N/2}$ from step 0 and $2^{2N/2}$ for g_{31}, h_{31}). Thanks to the filter, on average only $2^{5N/2}$ possibilities remains for step 2.

In step 2, there are $2^{6N/2}$ possibilities for the set of values $\{a_{17}, e_{17}, f_{17}, g_{17}, h_{17}, g_{31}, h_{31}, a_{34}\}$ ($2^{5N/2}$ from step 1 and $2^{N/2}$ for a_{34}). Thanks to the filter, on average only $2^{5N/2}$ possibilities remains for step 2.

In step 3 we consider $2^{5N/2} \times 2^{2N/2}$ possibilities, on average only $2^{6N/2}$ of them pass the filter.

In step 4 we consider $2^{6N/2}$ possibilities, on average only $2^{4N/2}$ of them pass the filter.

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

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

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 $x_{73}^{(2)} \oplus y_{73}^{(2)} = w_{73}^{(2)}$

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

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

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

We keep repeating these three steps (case a, case gh, and case 0) until we reach $n = 243$. 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 has 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.

In each step, there are never more than $2^{7N/2}$ possibilities tested (the maximum is in step 3). We can assume that the complexity is roughly $2^{7N/2}$. For $N = 8$, we obtain 2^{28}, which is coherent with our experimental results. For $N = 32$, it would give 112 bits of security, which is enough for NIST's standards, but far lower than the claim of 1024 bits of security.

Chapter 9

Conclusion and perspectives

The Linear Congruential Generator is at least seventy years old. As it is one of the oldest and best known pseudo-random numbers generator, it had been heavily studied and it was already known not to be cryptographically secure. All the efforts that were made to increase its security failed. As it is easy to implement, the LCG is a key part of several PRNGs, like the PCG presented in chapter 3 or Trifork in chapter 4, but these generators fail to hide the linear structure of the LCG enough and it is their main liability. We stripped all the layers that covered the LCG until we managed to obtain a truncated and possibly altered version of its outputs. From then all we had to do was to apply the already known attacks against the LCG. In chapter 5 we presented the Fast Knapsack generator that does not use an LCG. But we still managed to find a way to link them both which means all the work on the LCG could be used to attack this generator. We could conclude that using the LCG in a cryptographic environment not directly as a PRNG but even as a subroutine in another larger PRNG seems unwise.

In the second part we presented combined generators. Arrow and its predecessor Trifork combine two or more copies of a weak generator, the Lagged Fibonacci Generator, mixing them using binary operation to mask the linear structure. In the case of Trifork the weakness of the Lagged Fibonacci Generator allows us to easily attain the LCG used in the initialisation phase. In Arrow the weakness of the generator combined with the small size of the words in the internal state create a perfect set up to run guess-and-determine attacks. The problem is not restrained to the LCG: topping permutations or binary operations over a weak PRNG does not strengthen their security enough to make them suitable for cryptographic purpose.

The Knapsack Generator is a PRNG that combines a weak PRNG, a Linear Feedback Shift Register, and a hard computational problem, the Subset Sum Problem. The attacks both against the Knapsack Generator and its Elliptic version in chapter 7 start the same way by guessing the whole key of the LFSR. Because the Subset Sum Problem is a hard problem (even harder when the weights are secret), the outputs of the LFSR are not easily attainable. But in the set of parameters proposed in both papers presenting the Knapsack Generator and the Elliptic Knapsack Generator, the seed of the LFSR is small enough to be practically guessed (32 bits for the first PRNG, 16 for the second).

In chapter 7 we presented a generalised version of the Knapsack Generator and called the first weak generator generating the u_i's the *Control Sequence Generator* (CSG). It could be an LFSR (like in the original knapsack generator) or anything else. To make it secure, what properties should

this CSG satisfy? Is “having a seed large enough not to be guessed” enough? Could it lead to a secure version of the knapsack generator under the assumption that the Subset Sum is a hard problem? It would be more efficient than the Naor-Impagliazzio scheme [31] which is secure under this hypothesis but not efficient.

Bibliography

[1] Ahmadi, O., Shparlinski, I.E.: Exponential sums over points of elliptic curves. J. Number Theory **140**, 299–313 (2014)

[2] Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 717–746. Springer, Heidelberg (May 2019). doi:10.1007/978-3-030-17656-3_25

[3] Arratia, R., Gordon, L.: Tutorial on large deviations for the binomial distribution. Bulletin of Mathematical Biology **51**(1), 125 – 131 (1989)

[4] Babai, L.: On lovász'lattice reduction and the nearest lattice point problem. Combinatorica **6**, 1–13 (1986)

[5] 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). doi:10.1007/978-3-540-39887-5_10

[6] Banegas, G.: Attacks in stream ciphers: A survey. Cryptology ePrint Archive, Report 2014/677 (2014), https://eprint.iacr.org/2014/677

[7] Barak, B., Halevi, S.: A model and architecture for pseudo-random generation with applications to /dev/random. In: Atluri, V., Meadows, C., Juels, A. (eds.) ACM CCS 2005. pp. 203–212. ACM Press (Nov 2005). doi:10.1145/1102120.1102148

[8] Bauer, A., Vergnaud, D., Zapalowicz, J.C.: Inferring sequences produced by nonlinear pseudorandom number generators using Coppersmith's methods. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 609–626. Springer, Heidelberg (May 2012). doi:10.1007/978-3-642-30057-8_36

[9] Benhamouda, F., Chevalier, C., Thillard, A., Vergnaud, D.: Easing Coppersmith methods using analytic combinatorics: Applications to public-key cryptography with weak pseudorandomness. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part II. LNCS, vol. 9615, pp. 36–66. Springer, Heidelberg (Mar 2016). doi:10.1007/978-3-662-49387-8_3

[10] Blackburn, S.R., Ostafe, A., Shparlinski, I.E.: On the distribution of the subset sum pseudorandom number generator on elliptic curves. Unif. Distrib. Theory **6**(1), 127–142 (2011)

[11] Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic curves in cryptography, Lond. Math. Soc. Lect. Note Ser., vol. 265. Cambridge: Cambridge University Press (1999)

[12] 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)

[13] Bleichenbacher, D., Patel, S.: SOBER crytanalysis. In: Knudsen, L.R. (ed.) FSE'99. LNCS, vol. 1636, pp. 305–316. Springer, Heidelberg (Mar 1999). doi:10.1007/3-540-48519-8_22

[14] Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., Zimmermann, P.: Comparing the difficulty of factorization and discrete logarithm: A 240-digit experiment. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 62–91. Springer, Heidelberg (Aug 2020). doi:10.1007/978-3-030-56880-1_3

[15] Bouillaguet, C., Martinez, F., Sauvage, J.: Practical seed-recovery for the PCG pseudo-random number generator. IACR Trans. Symm. Cryptol. **2020**(3), 175–196 (2020). doi:10.13154/tosc.v2020.i3.175-196

[16] Cattaneo, G., De Maio, G., Petrillo, U.F.: Security issues and attacks on the gsm standard: a review. J. Univers. Comput. Sci. **19**(16), 2437–2452 (2013)

[17] Contini, S., Shparlinski, I.: On Stern's attack against secret truncated linear congruential generators. In: Boyd, C., Nieto, J.M.G. (eds.) ACISP 05. LNCS, vol. 3574, pp. 52–60. Springer, Heidelberg (Jul 2005)

[18] Coppersmith, D.: Finding a small root of a bivariate integer equation; factoring with high bits known. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 178–189. Springer, Heidelberg (May 1996). doi:10.1007/3-540-68339-9_16

[19] Coppersmith, D.: Finding a small root of a univariate modular equation. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 155–165. Springer, Heidelberg (May 1996). doi:10.1007/3-540-68339-9_14

[20] Coster, M.J., Joux, A., LaMacchia, B.A., Odlyzko, A.M., Schnorr, C.P., Stern, J.: Improved low-density subset sum algorithms. Computational complexity **2**, 111–128 (1992)

[21] Coveyou, R.: Serial correlation in the generation of pseudo-random numbers. Journal of the ACM (JACM) **7**(1), 72–74 (1960)

[22] Diem, C.: On the discrete logarithm problem in elliptic curves. Compos. Math. **147**(1), 75–104 (2011)

[23] Dodis, Y., Pointcheval, D., Ruhault, S., Vergnaud, D., Wichs, D.: Security analysis of pseudo-random number generators with input: /dev/random is not robust. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 647–658. ACM Press (Nov 2013). doi:10.1145/2508859.2516653

[24] El-Mahassni, E.D.: On the distribution of the elliptic subset sum generator of pseudorandom numbers. Integers **8**(1), article a31, 7 (2008)

[25] Ferrenberg, A.M., Landau, D.P., Wong, Y.J.: Monte carlo simulations: Hidden errors from "good" random number generators. Phys. Rev. Lett. **69**, 3382–3384 (Dec 1992). doi:10.1103/PhysRevLett.69.3382, `https://link.aps.org/doi/10.1103/PhysRevLett.69.3382`

[26] Franke, D.: How I hacked hacker news (with arc security advisory). `https://news.ycombinator.com/item?id=639976` (2009)

[27] Frieze, A.M., Hastad, J., Kannan, R., Lagarias, J.C., Shamir, A.: Reconstructing truncated integer variables satisfying linear congruences. SIAM J. Comput. **17**(2), 262–280 (Apr 1988). doi:10.1137/0217016, `http://dx.doi.org/10.1137/0217016`

[28] Von zur Gathen, J., Shparlinski, I.E.: Subset sum pseudorandom numbers: fast generation and distribution. Journal of Mathematical Cryptology **3**(2), 149–163 (2009)

[29] Golic, J.D.: Cryptanalysis of alleged A5 stream cipher. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 239–255. Springer, Heidelberg (May 1997). doi:10.1007/3-540-69053-0_17

[30] Herrmann, M., May, A.: Attacking power generators using unravelled linearization: When do we output too much? In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 487–504. Springer, Heidelberg (Dec 2009). doi:10.1007/978-3-642-10366-7_29

[31] Impagliazzo, R., Naor, M.: Efficient cryptographic schemes provably as secure as subset sum. Journal of Cryptology **9**(4), 199–216 (Sep 1996). doi:10.1007/BF00189260

[32] Jochemsz, E., May, A.: A strategy for finding roots of multivariate polynomials with new applications in attacking RSA variants. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 267–282. Springer, Heidelberg (Dec 2006). doi:10.1007/11935230_18

[33] Kaas, R., Buhrman, J.: Mean, median and mode in binomial distributions. Statistica Neerlandica **34**, 13–18 (1980)

[34] Keery A. McKay, L.B.: Report on lightweight cryptography. Tech. Rep. NISTIR 8114, National Institute of Standards and Technology, Gaithersburg, MD (2017). doi:10.6028/NIST.IR.8114

[35] Knellwolf, S., Meier, W.: Cryptanalysis of the knapsack generator. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 188–198. Springer, Heidelberg (Feb 2011). doi:10.1007/978-3-642-21702-9_11

[36] Knuth, D.: Deciphering a linear congruential encryption. IEEE Transactions on Information Theory **31**(1), 49–52 (1985)

[37] Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms. Addison-Wesley (1969)

[38] Knuth, D.E.: The art of computer programming, Volume 4B: Combinatorial Algorithms, Part 2. Addison-Wesley (2022)

[39] Lagarias, J.C., Odlyzko, A.M.: Solving low density subset sum problems. 24th Annual Symposium on Foundations of Computer Science (sfcs 1983) pp. 1–10 (1983)

[40] L'Ecuyer, P.: Combined multiple recursive random number generators. Operations research **44**(5), 816–822 (1996)

[41] Lehmer, D.H.: Mathematical methods in large-scale computing units. In: Proceedings of a Second Symposium on Large Scale Digital Calculating Machinery. pp. 141–151 (1949)

[42] Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische annalen **261**(ARTICLE), 515–534 (1982)

[43] 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)

[44] L'Écuyer, P.: Good parameters and implementations for combined multiple recursive random number generators. Operations Research **47**(1), 159–164 (1999)

[45] L'Écuyer, P.: Random number generation with multiple streams for sequential and parallel computing. In: 2015 Winter Simulation Conference (WSC). pp. 31–44. IEEE (2015)

[46] Marsaglia, G.: Random numbers fall mainly in the planes. Proceedings of the National Academy of sciences **61**(1), 25–28 (1968)

[47] Martinez, F.: Attacks on pseudo random number generators hiding a linear structure. In: Galbraith, S.D. (ed.) CT-RSA 2022. LNCS, vol. 13161, pp. 145–168. Springer, Heidelberg (Mar 2022). doi:10.1007/978-3-030-95312-6_7

[48] Martinez, F.: Practical seed-recovery of fast cryptographic pseudo-random number generators. In: Ateniese, G., Venturi, D. (eds.) ACNS 22. LNCS, vol. 13269, pp. 212–229. Springer, Heidelberg (Jun 2022). doi:10.1007/978-3-031-09234-3_11

[49] Mefenza, T., Vergnaud, D.: Inferring sequences produced by elliptic curve generators using coppersmith's methods. Theoretical Computer Science **830**, 20–42 (2020)

[50] Mitra, A.: On the properties of pseudo noise sequences with a simple proposal of randomness test. International Journal of Electrical and Computer Engineering **3**(3), 164–169 (2008)

[51] 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)

[52] Ritzenhofen, M.: On efficiently calculationg small solutions of systmes of polynomial equations: lattice-based methods and applications to cryptography. Ph.D. thesis, Verlag nicht ermittelbar (2010)

[53] 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). doi:10.1007/BFb0053728

[54] Ross, S.: Probability Models for Computer Science. Elsevier Science (2002), `https://books.google.fr/books?id=fG3iEZ8f3CcC`

[55] Rotenberg, A.: A new pseudo-random number generator. J. ACM **7**(1), 75–77 (jan 1960). doi:10.1145/321008.321019, https://doi.org/10.1145/321008.321019

[56] Rueppel, R.A., Massey, J.L.: Knapsack as a nonlinear fonction. In: IEEE International Symposium on Information Theory. IEEE Press, NY (1985)

[57] Semaev, I.: Summation polynomials and the discrete logarithm problem on elliptic curves. Cryptology ePrint Archive, Report 2004/031 (2004), https://eprint.iacr.org/2004/031

[58] Shoup, V.: Lower bounds for discrete logarithms and related problems. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 256–266. Springer, Heidelberg (May 1997). doi:10.1007/3-540-69053-0_18

[59] Stern, J.: Secret linear congruential generators are not cryptographically secure. In: 28th FOCS. pp. 421–426. IEEE Computer Society Press (Oct 1987). doi:10.1109/SFCS.1987.51

[60] Sun, H.Y., Zhu, X.Y., Zheng, Q.X.: Predicting truncated multiple recursive generators with unknown parameters. Designs, Codes and Cryptography **88**(6), 1083–1102 (2020)

[61] development team, T.F.: `fplll`, a lattice reduction library (2016), https://github.com/fplll/fplll, available at https://github.com/fplll/fplll

[62] Thomson, W.E.: A Modified Congruence Method of Generating Pseudo-random Numbers. The Computer Journal **1**(2), 83–83 (01 1958). doi:10.1093/comjnl/1.2.83, https://doi.org/10.1093/comjnl/1.2.83

[63] Van der Walt, S., Colbert, S.C., Varoquaux, G.: The NumPy array: A structure for efficient numerical computation. Computing in Science Engineering **13**(2), 22–30 (2011)

[64] von zur Gathen, J., Shparlinski, I.: Predicting subset sum pseudorandom generators. In: Handschuh, H., Hasan, A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 241–251. Springer, Heidelberg (Aug 2004). doi:10.1007/978-3-540-30564-4_17

[65] Washington, L.C.: Elliptic curves. Number theory and cryptography. Boca Raton, FL: Chapman and Hall/CRC, 2nd ed. edn. (2008)

[66] Yao, H., Wornell, G.W.: Lattice-reduction-aided detectors for mimo communication systems. In: Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE. vol. 1, pp. 424–428. IEEE (2002)

[67] Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In: Feldmann, A., Mathy, L. (eds.) Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009. pp. 15–27. ACM (2009). doi:10.1145/1644893.1644896, https://doi.org/10.1145/1644893.1644896

[68] Yu, H.B., Zheng, Q.X., Liu, Y.J., Bi, J.G., Duan, Y.F., Xue, J.W., Wu, Y., Cao, Y., Cheng, R., Wang, L., et al.: An improved method for predicting truncated multiple recursive generators with unknown parameters. Designs, Codes and Cryptography pp. 1–24 (2023)