FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Mykhailo Naumenko

Cryptographically Secure Random Number Generators

Department of Algebra

Supervisor of the master thesis:	Dr. rer. nat. Faruk Göloğlu
Study programme:	Mathematics
Study branch:	Mathematics for Information Technology

Prague 2024

This thesis is dedicated to Dr. rer. nat. Faruk Göloğlu, whose wisdom and knowledge have been my guiding light throughout this academic journey.

Title: Cryptographically Secure Random Number Generators

Author: Mykhailo Naumenko

Department: Department of Algebra

Supervisor: Dr. rer. nat. Faruk Göloğlu, Department of Algebra

Abstract: The thesis explores the theoretical and practical aspects of Cryptographically Secure Pseudorandom Number Generators (CSPRNGs) in modern cryptography and computer security. The study delves into theoretical background, the construction, security measures, and practical implementations of CSPRNGs, emphasizing their importance in secure communication channels and cryptographic protocols. Through an extensive literature review, this work highlights the challenges in achieving absolute security for pseudorandom number generators, establishing that they can be constructed from one-way functions and significantly expanded while maintaining security, provided.

The thesis also examines various well-known CSPRNG algorithms such as Yarrow, Fortuna, ChaCha20, ISAAC, ANSI X9.17, and others. The security features and known vulnerabilities are identified from available literary sources. Practical attacks on these generators, including state compromise, chosen-input attacks, and backtracking, are analyzed to underscore the importance of robust design and proactive security measures.

Moreover, the study presents practical implementations of unsecure algorithms in programming environments, showcasing their application and potential weaknesses in real-world scenarios, emphasizing that only verified means should be used for practical implementations. By analyzing historical and contemporary attacks, the research underscores the necessity for continuous improvements in PRNG designs to safeguard against evolving threats.

Keywords: Cryptographically Secure Random Number Generators (CSPRNG), Pseudorandom Number Generators (PRNG), Yarrow Algorithm, Fortuna Algorithm, Cryptographic Attacks, Entropy Accumulation, Stream Ciphers

Contents

Introduction

Cryptographically secure pseudorandom number generators (CSPRNG) are commonly used in modern cryptography and computer security applications. The main motivation behind the study of CSPRNGs is to generate unpredictable and unbiased sequences of numbers that are suitable for cryptographic purposes. Cryptographic security requires that the generated numbers are indistinguishable from truly random numbers, and that no information about the internal state of the generator can be inferred from the output.

This thesis explores the theoretical foundations of CSPRNGs, focusing on their construction and analysis. As such, the mathematical properties of Cryptographically Secure PRNGs and the conditions under which a generator can be considered cryptographically secure will be investigated.

The main goal of the thesis is research on available implementations of Pseudorandom Number Generators in different programming languages and computer systems, and to explore known practical attacks on generators previously deemed secure. We will also develop an example of the weakness of the system endangered by the use of cryptographically insecure Pseudorandom Number Generator.

1. Preliminaries

Before diving into the topic of CSPRNGs, several important mathematical concepts have to be described. As such, it will include:

- A negligible parameter,
- Asymptotic notation,
- Family of functions,
- Polynomial-time function,
- Computational hardness and complexity theory P, NP, EXP, etc.,
- Distribution ensemble (also reffered to as probability ensemble).

The above definitions are derived from [2].

Definition 1 (Negligible parameter)**.** Let ε be a positive real number. A parameter λ is said to be *negligible* if, for any positive integer k, there exists a positive integer N such that for all $n > N$, the following inequality holds:

$$|\lambda(n)| < \frac{1}{n^k},$$

where $\lambda(n)$ denotes the value of the parameter λ at input of size n.

Definition 2. The expectation of a random variable with a countably infinite set of possible outcomes is defined as the weighted average of all possible outcomes, where the weights are given by the probabilities of realizing each given value:

$$\mathrm{E}[X] = \sum_{i=1}^{\inf} X_i p_i.$$

Definition 3 (Asymptotic notation)**.** [2, p.5]
Let both $k(n)$ and $l(n)$ be positive numbers. We use the following notation:
$l(n) = \mathcal{O}(k(n))$ if there exists $c > 0$ such that $l(n) \leq c \cdot k(n)$,
$l(n) = o(k(n))$ if for all $c > 0$ holds that $l(n) \leq c \cdot k(n)$,
$l(n) = \Theta(k(n))$ if there exists $c_1, c_2 > 0$ such that $c_1 \cdot k(n) \leq l(n) \leq c_2 \cdot k(n)$.

Definition 4 (Polynomial parameter)**.** [2, p.6]
Assuming that $n \in \mathbb{N}$, number $l(n)$ is a polynomial parameter if $l(n) = n^{\mathcal{O}(1)}$ and $l(n)$ is computable in time $n^{\mathcal{O}(1)}$[2, p.6].

Definition 5 (Family of functions)**.** [2, p.6]
We call $f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ a *family of functions* if for all n function f_n can be computed by the same algorithm parameterised by n.

Generally and informally speaking, every function performs the same transformation for inputs of different sizes.

Family of functions serves as a way to formally denote functions with variable lengths of input and output.

Definition 6. Polynomial-time function Function $f : \{0,1\}^n \longrightarrow \{0,1\}^*$ is a *Polynomial-time function* if there exists a Turing machine A such that for all $x \in \{0,1\}^n$ it computes $f(x)$ in time $n^{\mathcal{O}(1)}$

Definition 7 (Computational hardness and complexity theory P, NP, EXP)**.** In computational complexity theory, a *complexity class* is a set of decision problems that share a common level of computational difficulty. Complexity classes provide a framework for classifying problems based on the resources required to solve them algorithmically, such as time or space. There is a common notation for general computation classes. **P** denotes the set of decision problems that can be solved by a deterministic Turing machine in polynomial time. A language $L \in \mathbf{P}$ if there exists a polynomial-time algorithm A such that for every input x, A correctly decides if $x \in L$.

NP denotes the set of decision problems for which a solution can be verified in polynomial time. A language $L \in \mathbf{NP}$ if there exists an algorithm A and a polynomial-time algorithm (verifier) V such that for every input x and a purported by A solution y, V can determine whether y is a valid solution for x.

EXPTIME denotes set of decision problems that can be solved by a deterministic Turing machine in exponential time. A language $L \in \mathbf{EXPTIME}$ if there exists an exponential-time algorithm A such that for every input x, A correctly decides whether $x \in L$.

Definition 8 (distribution ensemble)**.** A *distribution ensemble* or probability ensemble isfamily of distributions $\mathcal{D}_n$, where index n is a natural number and every distribution $\mathcal{D}_n$ has similar characteristics for n sufficiently large.

Definition 9 (Expected value)**.** [16, p.9]

The expected value of a random variable with a countably infinite set of possible outcomes is defined as:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} x_i \Pr[X = x_i].$$

Lattices

To understand (CSPRNGs), we must explore key mathematical structures. This leads us to lattices and the LLL algorithm, which are crucial in cryptography and widely used in cryptanalysis to test system security.

Definition 10 (Lattice)**.** [12, 13] A *lattice* L in $\mathbb{R}^n$ is a discrete subset of $\mathbb{R}^n$ that can be expressed as

$$L = \left\{ \sum_{i=1}^{k} a_i \mathbf{b}_i \mid a_i \in \mathbb{Z}, \mathbf{b}_i \in \mathbb{R}^n \right\},$$

where $\{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k\}$ is a set of linearly independent vectors in $\mathbb{R}^n$, known as the *basis* of the lattice.

Let $V = (\mathbf{b}_1 | \mathbf{b}_2 | \cdots | \mathbf{b}_k)$ be the $n \times k$ matrix whose columns are $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k$, then determinant of lattice L is defined as

$$\det(L) = \sqrt{\det(V^T V)}.$$

The *Shortest Vector Problem* (SVP) in lattice theory is the computational problem of finding the shortest non-zero vector in a lattice L, i.e., solving for

$$\mathbf{v}_{\min} = \arg \min_{\mathbf{v} \in L \setminus \{\mathbf{0}\}} \|\mathbf{v}\|,$$

where $\| \cdot \|$ denotes the Euclidean norm.

The i-th minimum of a lattice L, denoted as $\lambda_i(L)$, is the smallest value such that there are i linearly independent vectors in L with norms no greater than $\lambda_i(L)$. It represents the radius of the smallest ball that contains i linearly independent lattice vectors. Clearly, $\lambda_1(L)$ is the norm of the shortest non-zero vector in L.

The LLL algorithm, developed by A. Lenstra, H. Lenstra, and L. Lovász [12], is a polynomial-time algorithm for lattice basis reduction that approximates the solution to the Shortest Vector Problem (SVP). It transforms an arbitrary lattice basis into a reduced basis where vectors are shorter and more orthogonal, making the lattice easier to work with for various computational problems.

Definition 11. A basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$ for a lattice $L \subseteq \mathbb{R}^n$ is LLL-reduced if it satisfies the following conditions for a given parameter δ with $1/4 < \delta < 1$:

1. Size Reduction: $\forall 1 \leq j < k \leq n$ the coefficient of $\mathbf{b}_j$ in the Gram-Schmidt orthogonalization of $\mathbf{B}$ satisfies

$$|\mu_{k,j}| \leq \frac{1}{2},$$

where $\mu_{k,j} = \frac{\langle \mathbf{b}_k, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}^*{}_j, \mathbf{b}^*{}_j \rangle}$ and $\mathbf{b}_j^*$ are the Gram-Schmidt orthogonalized vectors.

2. Lovász Condition: For all $1 \leq k < n$,

$$\delta ||\mathbf{b}_k^*||^2 \leq ||\mathbf{b}_{k+1}^* + \mu_{k+1,k} \mathbf{b}_k^*||^2.$$

These conditions ensure that the vectors in the LLL-reduced basis are relatively short and nearly orthogonal to each other, which is useful for approximating the shortest vector in the lattice.

The reduction parameter δ is usually set to $\frac{3}{4}$, LLL algorithm is proved to run in polynomial time for $\delta = \frac{3}{4}$. The LLL algorithm guarantees that the first vector in the reduced basis is within a factor of $2^{(n-1)/2}$ of the shortest vector in the lattice, where n is the dimension of the lattice.

It is proved that for a δ-reduced basis $\{v_1, \ldots, v_n\}$ of n-dimensional lattice L the following holds true [6, Lemma 5] [12]:

1. $||v_i|| \leq (\frac{4}{4\delta - 1})^{(n-1)/2} \lambda_i(L)$;

2. $||v_i|| \leq (\frac{4}{4\delta - 1})^{(n-1)/4} \det(L)^{1/n}$.

Cryptographic Primitives

LFSR. A linear feedback shift register (LFSR) is a mathematical construct often used in cryptography and computer science. As a rule, this is a register with a finite sequence of binary bits $(x_0, x_1, \ldots, x_n)$, where n is the length of the register [25]. The state of the LFSR changes according to the following pattern: in discrete time steps, the entire register is shifted by one bit, resulting in one bit (say at the end of the register) being the output of the step. At the same time, one bit in the beginning of the register is empty after the shift. A new value for this bit is determined by the linear combination of the rest n bits including the output bit. This linear combination is defined by a fixed feedback polynomial that defines the LFSR taps that are used in the calculation. The resulting bit sequence generated by LFSR exhibits pseudo-random behavior, making it suitable for applications such as encryption key generation or pseudo-random number generation.

RC4 stream cipher. Stream cipher RC4 (Rivest Cipher 4) is a symmetric cipher previously used in cryptography but now no longer considered secure due to recovered attacks. It functions by generating a pseudorandom stream of bytes, which is then combined with the plaintext or ciphertext using the XOR operation. Its structure is very similar to LFSR except for one difference: RC4 operates with bytes, not bits, which makes it convenient for software implementation whereas LFSR is adapted for implementation on hardware.

Mathematically, RC4 can be formally described as follows:

- *Initialization*: The internal state of RC4 is 256 bytes, initialized with values from 0 to 255.
- *Key scheduling algorithm (KSA)*: KSA reorders the internal state array S based on the key. This is achieved by performing a series of exchanges. Swaps are defined by key bytes and the current state of C.
- *Pseudo-Random Generation Algorithm (PRGA)*: After KSA, RC4 enters the PRGA phase. During PRGA, bytes are generated one at a time from the state array S and output. This is done by updating the state array in a non-linear and deterministic way, guided by the current key byte and the state S.
- *Key Stream Generation*: The output of RC4 is a pseudo-random key stream that can be used for encryption or decryption. The key stream is obtained by reapplying PRGA, creating a sequence of bytes based on the updated state array S.

In general, RC4 uses the KSA and PRGA procedures to convert the secret key into a pseudo-random key stream. The generated keystream has the properties of randomness and is combined with plaintext or ciphertext using XOR for privacy and cryptographic operations.

Hash function. Hash functions are fundamental cryptographic algorithms that can be used for a variety of purposes. They accept inputs of any length and produce outputs of a fixed size. Hash functions are designed to be fast and efficient.

A cryptographic hash function h must have the following properties according to [1, pp. 322 – 324]:

- Deterministic: $x = y$ implies that $h(x) = h(y)$.
- Fast Computation: The hash function must be able to calculate the hash value "quickly", even for large inputs.
- Pre-image resistance: knowing $h(x)$ the task of finding x must be "inefficient", impossible in polynomial time.
- Collision resistant: It should be "hard" to find two different inputs $x_1 \neq x_2$ that produce the same hash value $h(x_1) = h(x_2)$.
- Avalanche effect: A small change in the input data should result in a significant change in the hash value.
- Uniformity: The hash function must evenly distribute the outputs across the output space. Assuming that h is a perfect hash function with an output of length $n \in \mathbb{N}$, it should hold that for $x_1 \neq x_2$ the probability $\Pr[[h(x_1)]_i = [h(x_2)]_i] = \frac{1}{2}$ for every $i \leq n$, where $[h(x)]_i$ denotes i-th bit of $h(x)$.

Examples of hash functions used in practice are: MD5, SHA-1, SHA-256, Blake2 and further.

Block ciphers. Block ciphers are cryptographic algorithms designed to encrypt and decrypt fixed-size blocks of data. They form a fundamental building block of modern symmetric-key encryption systems. In a block cipher, the plaintext is divided into fixed-size blocks, typically 64 or 128 bits, and each block is transformed into ciphertext using a specific encryption algorithm and a secret key. Block ciphers are usually constructed using the Feistel network structure, involving rounds of substitution, permutation, and key mixing operations. The security of block ciphers relies on strong secret keys generated through key schedules. Their construction combines mathematical operations, key management, and security considerations to enable secure and efficient encryption and decryption in cryptographic applications.

Examples of block ciphers include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).

Block ciphers can be used in various modes of operation to provide additional security features. In the description of the modes below, to simplify the notation, we denote i-th block of ciphertext and plaintext with C_i and P_i respectively, $E_K(x)$ denotes encryption with key K. $E_K^i(x)$ denotes encryption repeated i times[23].

- Counter mode (CTR) refers to the encryption mode when block of plaintext is combined with an encrypted initial vector (IV) using XOR, next block is combined with $IV + 1$ and so on.
  $C_i = P_i \oplus E_K(IV + i)$

- Electronic codebook (ECB) is a mode, where every block is encrypted separately without any interaction.
  $C_i = E_K(P_i)$
- Cipher block chaining (CBC). In CBC previous block of ciphertext is always added to the next block of plaintext before encryption.
  $C_i = E_K(P_i \oplus C_{i-1})$
- Propagating CBC (PCBC) mode works in the way where combination of previous block plaintext and ciphertext is always added to the next block of plaintext before encryption.
  $C_i = E_K(P_i \oplus P_{i-1} \oplus C_{i-1})$
- Cipher feedback (CFB). In CFB the next cipher block is created by addition of encrypted previous cipher block to the next block of plaintext.
  $C_i = E_K(C_{i-1}) \oplus P_i$
- Output feedback (OFB) mode operates in the following way. IV is encrypted n times to be combined with n-th block of plain text.
  $C_i = E^i_K(IV) \oplus P_i$

In the following table we compare modes of operation mentioned above.

Table 1.1: Encryption and Decryption Properties of Various Modes of Operation

Mode	ECB	CBC	CFB	OFB	CTR
Parallel Encryption	Y	N	N	N	Y
Parallel Decryption	Y	Y	Y	N	Y
Precomputation Possible	N	N	N	Y	Y
Error Propagation	1 block	1 block	1 block	N	N
Self-Synchronizing	Y	Y	Y	N	N
Encryption = Decryption	N	N	Y	Y	Y

The above table summarizes advantages and disadventages of each mode of operations. For example, ECB and CTR modes allow parallelization of encryption, meaning that multiple messages can be processed at the same time as theere is no dependence on previous encrypted block. Similarly, all mentioned modes but OFB allow parallelization of decryption.

As we will show later, some modes of operation of a block cipher can be considered a PRNG.

2. Theoretical foundations of CSPRNG

In this chapter we will dive into computer science background of pseodorandomness generators. We will start with important concepts required to define a PRNG. Then we will define a Cryptographically Secure PRNG and its most desired properties. We will state and prove that PRNG must be Next-bit unpredictable: this property can be used as a test of being a PRNG. In the rest of the chapter we will introduce theoretical constructions of PRNGs from one-way functions.

2.1 Introduction to PRNGs

This section outlines the concepts and terminology required to formally define a pseudo-random number generator (PRNG). As such, it introduces the following definitions:

- Distinguishing probability,
- Cryptographic security and distinguishability,
- Security parameter,
- Time-success ratio and further.

These definitions provide the necessary background for our subsequent discussion of CSPRNGs. This chapter is based entirely on Pseudorandomness and cryptographic applications by Michael Luby [2].

Definition 12 (Distinguishing probability)**.** [2, p. 15]
Let $A \in \{A_n : \{0,1\}^n \longrightarrow \{0,1\}\}$ be an instance of a family of functions, $n \in N$. Let $x, y \in \{0,1\}^n$ be uniformly chosen random inputs to A. The *distinguishing probability* of A for x and y is

$$\delta(n) = |\Pr_x[A(x) = 1] - \Pr_y[A(y) = 1]|.$$

Family of functions A in the definition above takes role of an adversary algorithm. Its outputs are interpreted as “reject” and “accept” The intention behind this definition is to measure a difference between sequences: it should accept every sequence that “looks like” x and reject every other sequence. This concept will lead us to the point when one can demonstrate that no adversary algorithm could tell apart deterministically generated string x and a genuinely random string y. The definition of distinguishing probability allows an informal definition of a PRNG:

Informal definition (Pseudorandom generator)**.** [2, p. 20]
Let $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be a function from a family of polynomial-time functions, where $l(n) > n$ is a polynomial parameter. Let x be a uniformly

random chosen input for f, let y be a uniformly randomly chosen from $\{0,1\}^{l(n)}$. Then we say that f is a PRNG if for every polynomial-time adversary A the distinguishing probability of a for $f(x)$ and y is "very small".

It is necessary to provide a rigorous definition of "very small" distinguishing probability to properly define the PRNG. The number $l(n)$ above plays role of a stretching parameter. It introduces the fact, that f as a PRNG usually only extends a random sequence to produce longer sequence. The difference $l(n) - n$ is usually referred to as a stretching parameter.

2.1.1 Security measure

Following the idea from [2], we will connect "very small" distinguishing probability with a measurement of security: it has to be small enough to satisfy the security needs. Let us introduce this and some other ancillary structures that serves to facilitate the later work.

Definition 13 (The security parameter)**.** [2, p. 24]
A *security parameter* $s(n)$ is a parameter that is used to determine the level of security of a cryptographic primitive. The larger the value of $s(n)$, the stronger the security.

In practice, the security parameter is usually determined by the size of the private memory of the computer that the cryptographic scheme utilizes. The private memory is the memory hidden by the scheme, and it is commonly the primary target of attackers. Typical example is the key. It is the most wanted information for every attacker.

By setting the security parameter to the size of the private memory, the scheme primes the attacker to perform a large number of computations in order to break the scheme, thus increasing its security. However, the security parameter is not always set to the size of the private memory, as other factors such as computational efficiency and practical considerations may also play a role in determining the appropriate value of $s(n)$.

An example where the security parameter is not equal to the size of the hidden key is DES. DES cipher has a 64-bit key, but it is referred to as "56-bit effective key"[1, p.252], because every 8-th bit is a parity bit (it can be computed as a sum of previous 7 bits). For this example, the size of private memory is 64 bits, but the security parameter $s(n)$ is only 56 as in a brute force attack, attacker would need to correctly guess "only" 56 bits.

When discussing attackers it's essential to establish their role in every cryptographic system. In cryptography, an adversary refers to an entity or a person who attempts to undermine or compromise the security of a system. Specifically, in the context of analyzing the security of cryptographic schemes, an adversary is a computational agent that has some degree of control over the inputs and outputs of the scheme, and aims to reveal the secret key or other confidential information.

Definition 14 (Adversary)**.** [2, p. 25]
the *adversary* is a function family that can be computed by a Turing machine. Adversary is a function that recovers secret and hence breaks the security

of the system. It is assumed to have unlimited computational power, unless explicitly stated otherwise. Two primary factors are used to evaluate the computational power of an adversary: the total time $T(n)$ the adversary spends on computation, and the probability of success $\delta(n)$ of the adversary in breaking the security of the system.

Throughout this thesis, we will refer to several types of adversaries. To simplify the perception of the text, allow us to indicate the notation in advance.

- *Inverting* adversary A for function f is an adversary that computes an element of preimage of $f(x)$. The notation is: $A(f(x))$. It is expected that for correct output of such adversary holds: $A(f(x)) \in f^{-1}(f(x))$, for arbitrary input x of function f ($f^{-1}(f(x))$ denotes the preimage).

- *Distinguishing* adversary A for x and y ($x, y \in \{0,1\}^n$, additionally x is an element of subset of $\{0,1\}^n$ with some Property) is an adversary that distinguishes an origin of the input. It returns just one of two possible messages: "accept(1)" if input appears to have the Property like x does, and "reject(0)" otherwise.

- *Next-bit predicting* adversary A for function f is an algorithm that tends to predict an output $f(x)$ on position i operating with all previous bits on positions (1 to $i-1$). The syntax for such adversary is

  $$A(i, f(x)_{\{1,2,\ldots,i-1\}}),$$

  where x is an arbitrary input of f, i denotes output position that will be predicted and f$(x)_{\{1,2,\ldots,i-1\}}$ denotes a substring of $f(x)$ on positions 1 to $i-1$.

As stated in definition, adversary is characterized by the computation time $T(n)$ and success probability $\delta(n)$. The following definition combines these aspects to define a single performance measure.

Definition 15 (Time-success ratio)**.** [2, p. 25]
Time-success ratio of an adversary A for a primitive f is

$$R(s(n)) = T(n)/\delta(n),$$

where $T(n)$ is the worst case time bound of A, $\delta(n)$ is the success probability of A for f and $s(n)$ is the security parameter of f.

Although it might be intuitive to use an expected value of time-success ratio when speaking about effectiveness of adversary, according to [2], the worst case time bound is still effective for our purposes.

Definition 16 (Security)**.** a primitive f is $S(s(n))$-secure if every adversary A for f has time-success ratio $R(s(n)) \geq S(s(n))$.

This framework and security definition is essential to present a realistic case of nondeterministic adversary. Following an example in [2, p. 26], there exists a natural way to trade off time complexity for a chance to success.

2.1.2 Reductions

A reduction is an algorithm for transforming one problem into another. Intuitively, reduction uses an algorithm that solves first problem as a subroutine to solve target problem.

Definition 17 (Oracle adversary)**.** Oracle adversary S is defined as an algorithm which is given an "oracle access" to another algorithm and exploits its outputs. Notation S^A denotes that oracle adversary S queries adversary A and uses answers (outputs) of A for own computations. The run time of S^A includes the time for computing A in the oracle calls to A.

To stay compatible with other sources terminology, we should mention that S^A would typically be interpreted as "adversary S that queries oracle A". It is still the same thing here: S is called oracle adversary and A is called adversary because A's task outside of our oracle scheme is to attack some primitive.

Oracle queries are important part of the uniform reduction which is defined below.

Definition 18 (Uniform reduction)**.** A "*uniform reduction*" is a polynomial-time algorithm that can transform instances of one problem into instances of another problem in a way that preserves the complexity of the problems. Specifically, we say that there is a uniform reduction from Primitive 1 to Primitive 2 if there exists polynomial-time adversary oracle P and adversary oracle S that satisfies two properties:

- for every instance f of Primitive 1 with security parameter $s(n)$ it holds that polynomial-time adversary oracle $g = P^f$ is inherently an instance of Primitive 2 with security parameter $s'(n)$,
- given any adversary A for g with Time-success ratio $R'(s'(n))$, oracle adversary S^A is an adversary for f with Time-success ratio $R(s(n))$.

Moreover, a uniform reduction must map inputs to inputs that are chosen uniformly at random.

Security Preserving

The effectiveness of a reduction from f to g is determined by how much of the security of f is retained in g. To measure this accurately, we compare the success rates of f and g when they use the same amount of private information. Reductions can be classified as linear-preserving, poly-preserving, or weak-preserving to give a rough measure of their security-preserving capabilities.

Definition 19 (reduction security)**.** Let f and g be polynomial-time function families both having the same security parameter (size of used private memory is N) and there is a reduction from f to g. Let us denote Adversary A for function family f has a time-success ratio $R(N)$, and reduction oracle adversary B for g has a Time-success ratio $R'(N)$.

We call reduction from f to g:

- linear-preserving if $R(N) = N^{\mathcal{O}(1)} \cdot \mathcal{O}(R'(N))$,

- poly-preserving if $R(N) = N^{\mathcal{O}(1)} \cdot (R'(N))^{\mathcal{O}(1)}$,
- weak-preserving if $R(N) = N^{\mathcal{O}(1)} \cdot (R'(N^{\mathcal{O}(1)}))^{\mathcal{O}(1)}$.

2.1.3 One-way functions

At this point, a closer look at one-way functions can be taken.

Definition 20 (One-way function)**.** [2, p.27]
Let $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be an instance of a family of polynomial-time functions with security parameter $s(n) = n$. Let x be a uniformly random element of set $\{0,1\}^n$. Success probability of an inverting adversary [14] A for f is

$$\delta(n) = \Pr[f(A(f(x))) = f(x)].$$

We say that f is a $S(n)$**-secure one-way function** if every adversary has time-success ratio greater than $S(n)$.

Alternatively, one-way function can be also defined in the following manner.

Definition 21. A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ is called a (strong) one-way function if it can be computed by deterministic algorithm A in a polynomial-time, and every randomised polynomial-time algorithm A' succeeds in inverting f for a negligible parameter $\lambda(n)$ with probability:

$$\Pr[A'(f(x)) \in f^{-1}(f(x))] < \lambda(n),$$

where n is sufficiently large natural number, and x is an arbitrary binary string of length n [16, definition 2.2.1].

To review the concordance of above definitions one can compare them and verify, that strong one-way function as in definition 21 corresponds to $(\frac{T'(n)}{\lambda(n)})$-secure one-way function in terms of definition 20 (assuming adversary A' has run-time $T'(n)$).

It is important to hold in mind, that typical assumption on the security level is such, that every cryptographical primitive is expected to be compromised in at least exponential time. Meaning that in practise we want to use $2^{\Theta(n)}$-secure primitives and stronger. In the above definitions this assumption is hidden behind the strict inequality that must hold for every positive polynomial n^k : it means that adversary's complexity is expected to be worse than polynomial. From definition 1, $\lambda(n)$ is negligible if for any positive integer k there exists a positive integer N such that for all $n > N$ it holds that $|\lambda(n)| < \frac{1}{n^k}$.

PRNGs exist only under assumption $\mathbf{P} \neq \mathbf{NP}$. So, formally, we do not know if any PRNG exist in practice. Same holds for one way functions:

Claim 1. $\mathbf{P} = \mathbf{NP}$ *implies that one-way functions do not exist.*

Proof. Let f be a one-way function. Then by definition, it's inverse function f^{-1} is in **NP** complexity class, as f can be computed in the polynomial-time to verify the output produced by f^{-1}.

But by assumption $\mathbf{P} = \mathbf{NP}$. So, f^{-1} is also in **P**. Hence, an inverse of f can be solved in polynomial time by Turing machine with success probability 1.

□

With the introduction of one-way functions, it is natural to consider their benefits and potential uses. A common method to increase the security of cryptographic algorithms is to use reduction techniques to transform the security of one problem into that of another. In the context of a CSPRNG, these methods can be used to convert the unpredictability of one-way functions into the unpredictability of random sequences, providing stronger cryptographic security guarantees. In another words, we may use reductions to transform one-way function and create a CSPRNG.

2.2 Cryptographically Secure PseudoRandom Number Generators

In this section, we will formally define a cryptographically secure pseudo-random number generator (CSPRNG). We will start by presenting the basic construction of a CSPRNG and explaining its properties. We will then introduce the concept of security for a CSPRNG and discuss the requirements that a generator must satisfy in order to be considered cryptographically secure.

Finally, we will state and prove the Next-bit Unpredictable Theorem [14, theorem 6.2.1], which shows that the existence of a one-way function implies the existence of a CSPRNG. This theorem, therefore, serves as a fundamental outcome in the theory of CSPRNGs and provides a strong connection between one-way functions and the design of secure generators.

Definition of CSPRNG A CSPRNG is a deterministic algorithm that produces a sequence of pseudo-random numbers as output. An input string that initializes the pseudorandom number generator defining its entire number sequence is called a random seed. If the generator is reset with an identical seed, it will reproduce the exact same sequence of numbers. The generator must satisfy the following properties:

- Next-bit unpredictability: Given the first k bits of the output, it should be computationally unfeasible to predict the $(k+1)$-th bit with high probability.
- Distinguishability from truly random sequences: the output sequence of the generator should be indistinguishable from a truly random sequence, even to an attacker with unbounded computational power.

Definition 22 (pseudorandom number generator)**.** [2, p. 50]

Let $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be an instance of a family of polynomial-time functions with security parameter $s(n) = n$. Let $l(n) > n$. Let x be a uniformly random element of a set $\{0,1\}^n$, z be a uniformly random element of a set $\{0,1\}^{l(n)}$. For every distinguishing adversary [14] A the distinguishing probability for f is

$$\delta(n) = \Big| \Pr[A(f(x)) = 1] - \Pr[A(z) = 1] \Big|.$$

Then, f is a *$S(n)$-secure pseudorandom generator* if every adversary has time-success ratio greater then $S(n)$.

An important property that we want pseudorandom number generator to have is a next-bit unpredictability. It is needed because should adversary be able to predict a sequence, it will be distinguishable from a random one.

Definition 23 (Next-bit unpredictable functions)**.** [2, p. 51]
Let $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be an instance of a family of polynomial-time functions with security parameter $s(n) = n$. Let $l(n) > n$. Let x be a uniformly random element of set $\{0,1\}^n$, i be a uniformly random index from set $\{1, 2, \ldots, l(n)\}$. The prediction probability of next-bit predicting adversary [14] A for f is

$$\delta(n) = \Big| \Pr[A(i, f(x)_{\{1,2,\ldots,i-1\}}) = f(x)_{\{i\}}] - \frac{1}{2} \Big|.$$

Then, f is a *$S(n)$-secure next-bit unpredictable* if every adversary has time-success ratio greater than $S(n)$.

Notice, that if the adversary always guesses the wrong bit, it can return as a result the opposite bit. Hence, the worst adversary's performance indicator is success probability of 1/2.

Theorem 2 (**PRNG $\Leftrightarrow$ Next-bit unpredictable (6.2.1 [14]**)**.** *Let function $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be an instance of a family of polynomial-time functions with security parameter $s(n) = n$, where $l(n) > n$. It holds that f is PRNG if and only if f is next-bit unpredictable. The reduction is linear-preserving in both directions.*

Proof.
"$\Longrightarrow$"

Assume, that A is a next-bit predicting adversary for the function f. So, $A(i, f(x)_{\{1,\ldots,i-1\}}) = f(x)_i$ with the probability

$$\delta'(n) = \Big| \Pr[A(i, f(x)_{\{1,2,\ldots,i-1\}}) = f(x)_{\{i\}}] - \frac{1}{2} \Big|.$$

A has run-time $T'(n)$ and Time-success ratio $R'(n) = \frac{T'(n)}{\delta'(n)}$.

Construct oracle adversary S^A for f in the sense of pseudorandomness in the following way. For input $z \in \{0,1\}^{l(n)}$ oracle S^A sends query to A to compute next-bit prediction *pred* on input $z_{\{1,\ldots,i-1\}}$ and compares it with z_i. Oracle is described in Algorithm 1.

Algorithm 1 Oracle S^A

```
Input: $z \in \{0,1\}^{l(n)}$, $i \leq l(n)$.
Output: 1 if $z$ is pseudorandom, 0 otherwise.
1: pred ← $A(i, z_{\{1,2,\ldots,i-1\}})$
2: if pred = $z_i$ then   return 1
3: else  return 0
4: end if
```

Probability, that S^A accepts input $f(x)$ (outputs 1) for random x is $\delta'(n)$. At the same time, the probability that S^A accepts a random string is $\frac{1}{2}$, because A cannot guess the next bit of truly random sequence.

Hence, in the sense of distinction, S^A has the following probability of success:

$$\delta(n) = \left|\delta'(n) - \frac{1}{2}\right|.$$

S^A needs only 1 query, so run time $T(n) = T'(n) + \mathcal{O}(1)$. Hence, the Time-success ratio is:

$$\begin{aligned}\frac{T(n)}{\delta(n)} &= \frac{T'(n) + \mathcal{O}(1)}{\left|\delta'(n) - \frac{1}{2}\right|}\\ &= \frac{T'(n) + \mathcal{O}(1)}{\delta'(n) + \mathcal{O}(1)}\\ &= \mathcal{O}\Big(\frac{T'(n)}{\delta'(n)}\Big).\end{aligned}$$

◪

"$\Longleftarrow$"

Assume, that A is a distinguishing adversary for f. It's probability of success is defined for an arbitrary input of f, $x \in \{0,1\}^n$, and for an uniformly random sequence of the same size as the output of f, $y \in \{0,1\}^{l(n)}$, as

$$\delta'(n) = \Big|\Pr[A(f(x)) = 1] - \Pr[A(y) = 1]\Big|.$$

WLoG assume that $\Pr[A(f(x)) = 1] > \Pr[A(y) = 1]$ holds always (there always exists NOT-A that returns opposite output):

$$\delta'(n) = \Pr[A(f(x)) = 1] - \Pr[A(y) = 1].$$

A has run-time $T'(n)$ and Time-success ratio $R'(n) = \frac{T'(n)}{\delta'(n)}$.

Let $x \in \{0,1\}^n$ be uniformly random input to f, let $y \in \{0,1\}^{l(n)}$ be uniformly random string. Now consider the sequence of strings that combine pseudorandom string with truly random string:

$$\begin{aligned}\mathcal{D}_0 &= (y_1, y_2, \dots, y_{l(n)}), && [= y]\\ \mathcal{D}_1 &= (f(x)_1, y_2, \dots, y_{l(n)}),\\ &\vdots\\ \mathcal{D}_i &= (f(x)_1, f(x)_2, \dots, f(x)_i, y_{i+1}, \dots, y_{l(n)}),\\ &\vdots\\ \mathcal{D}_{l(n)} &= (f(x)_1, f(x)_2, \dots, f(x)_{l(n)}). && [= f(x)]\end{aligned}$$

Now we consider the probabilities of success of A with any of the strings D_i. We use a telescoping sum to get the inequality. One can notice that the following is true regarding the probabilities of the first and last events in the sequence. $\Pr[A(\mathcal{D}_{l(n)}) = 1] = \Pr[A(f(x)) = 1]$ and $\Pr[A(\mathcal{D}_0) = 1] = \Pr[A(y) = 1]$. Hence,

$$\Pr[A(\mathcal{D}_{l(n)}) = 1] - \Pr[A(\mathcal{D}_0) = 1] = \delta'(n).$$

That implies that the mean value of difference is

$$\mathbf{E}\Bigg[\Pr[A(\mathcal{D}_i) = 1] - \Pr[A(\mathcal{D}_{i-1}) = 1]\Bigg] = \frac{\delta'(n)}{l(n)}.$$

Construct oracle adversary S^A for f in the sense of predictability in the following way.

Algorithm 2 Oracle S^A

```
Input: integer $i$, $(i-1)$-bits long string $z$
1: choose uniformly in random $(u_i, \dots, u_{l(n)})$
2: $b \leftarrow A(z_1, \dots, z_{i-1}, u_i, \dots, u_{l(n)})$
3: if $b = 1$ then return $u_i$
4: else return not $u_i$
5: end if
```

Now let us consider the probability of success for S^A. Let Q_{i-1} denote random variable of $S^A(z_1, \dots, z_{i-1}) = z_i$, so that Q_{i-1} equals 1 if the above equality holds and 0 otherwise. Let $\widetilde{z_i}$ denote a bit opposite to z_i. Note that $\Pr[u_i = z_i] = \frac{1}{2}$ as u_i is chosen uniformly random.

$$\begin{aligned}
\Pr[Q_{i-1}] &= \Pr[Q_{i-1}|u_i = z_i]\Pr[u_i = z_i] + \Pr[Q_{i-1}|u_i \neq z_i]\Pr[u_i \neq z_i] && (2.1)\\
&= \frac{1}{2}\Pr[b = 1|u_i = z_i] + \frac{1}{2}\Pr[b = 0|u_i \neq z_i] && (2.2)\\
&= \frac{1}{2}\Pr[A(z_1, \dots, z_{i-1}, z_i, u_{i+1}, \dots, u_{l(n)}) = 1] && (2.3)\\
&\quad + \frac{1}{2}\Pr[A(z_1, \dots, z_{i-1}, \widetilde{z_i}, u_{i+1}, \dots, u_{l(n)}) = 0]). && (2.4)
\end{aligned}$$

Let us analyse probability that A rejects input $(z_1, \dots, z_{i-1}, u_i, \dots, u_{l(n)})$ (in the algorithm above, $A(z_1, \dots, z_{i-1}, u_i, \dots, u_{l(n)})$ is assigned into variable b, hence we can shorten the notation of the probability to $\Pr[b = 0]$):

$$\begin{aligned}
\frac{1}{2}\Pr[b = 0|u_i \neq z_i] + \frac{1}{2}\Pr[b = 0|u_i = z_i] &= \Pr[b = 0], && (2.5)\\
\Pr[b = 0] &= 1 - \Pr[b = 1]. && (2.6)
\end{aligned}$$

At the same time, the following holds:

- $\Pr[b = 0|u_i \neq z_i] = \Pr[A(z_1, \dots, z_{i-1}, \widetilde{z_i}, u_{i+1}, \dots, u_{l(n)}) = 0]$,
- $\Pr[b = 0|u_i = z_i] = \Pr[A(z_1, \dots, z_{i-1}, z_i, u_{i+1}, \dots, u_{l(n)}) = 0]$
  $= \Pr[A(\mathcal{D}_i) = 0] = 1 - \Pr[A(\mathcal{D}_i) = 1]$,
- $\Pr[b = 1] = \Pr[A(z_1, \dots, z_{i-1}, u_i, u_{i+1}, \dots, u_{l(n)}) = 1]$
  $= \Pr[A(\mathcal{D}_{i-1}) = 1]$.

Substituting the above equalities into equation (2.5) we obtain:

$$\begin{aligned}\frac{1}{2}\Pr[A(z_1,\dots,z_{i-1},\widetilde{z}_i,u_{i+1},\dots,u_{l(n)})=0]+\frac{1}{2}(1-\Pr[A(\mathcal{D}_i)=1])\\ =1-\Pr[A(\mathcal{D}_{i-1})=1].\end{aligned}$$

Finally, by rearranging we get:

$$\begin{aligned}\frac{1}{2}\Pr[A(z_1,\dots,z_{i-1},\widetilde{z}_i,u_{i+1},\dots,u_{l(n)})=0]=\\ =1-\Pr[A(\mathcal{D}_{i-1})=1]-\frac{1}{2}(1-\Pr[A(\mathcal{D}_i)=1])\end{aligned}$$

We can now use these results in (2.1):

$$\begin{aligned}\Pr[Q_{i-1}] &= \frac{1}{2}\Pr[A(z_1,\dots,z_{i-1},z_i,u_{i+1},\dots,u_{l(n)})=1]\\ &\quad+\frac{1}{2}\Pr[A(z_1,\dots,z_{i-1},\widetilde{z}_i,u_{i+1},\dots,u_{l(n)})=0])\\ &=\frac{1}{2}\Pr[A(\mathcal{D}_i)=1]\\ &\quad+1-\Pr[A(\mathcal{D}_{i-1})=1]-\frac{1}{2}(1-\Pr[A(\mathcal{D}_i)=1])\\ &=\frac{1}{2}\Pr[A(\mathcal{D}_i)=1]+1-\Pr[A(\mathcal{D}_{i-1})=1]-\frac{1}{2}+\frac{1}{2}\Pr[A(\mathcal{D}_i)=1]\\ &=\frac{1}{2}+Pr[A(\mathcal{D}_i)=1]-\Pr[A(\mathcal{D}_{i-1})=1].\end{aligned}$$

Hence, on average it holds

$$\begin{aligned}\Pr[S^A(z_1,\dots,z_{i-1})=z_i] &= \frac{1}{2}+Pr[A(\mathcal{D}_i)=1]-\Pr[A(\mathcal{D}_{i-1})=1]\\ &=\frac{1}{2}+\frac{\delta'(n)}{l(n)}.\end{aligned}$$

To conclude, adversary S^A has run-time $T'(n)+\mathcal{O}(1)$ and success probability $\delta(n)=\frac{1}{2}+\frac{\delta'(n)}{l(n)}$. Hence, its Time-success ratio $R(n)$ is

$$\begin{aligned}\frac{T(n)}{\delta(n)} &= \frac{T'(n)+\mathcal{O}(1)}{\frac{1}{2}+\frac{\delta'(n)}{l(n)}}\\ &=\frac{T'(n)+\mathcal{O}(1)}{\mathcal{O}(\frac{\delta'(n)}{l(n)})}\\ &=\mathcal{O}(l(n))\times\frac{\mathcal{O}(T'(n))}{\mathcal{O}(\delta'(n))}\\ &=n^{\mathcal{O}(1)}\mathcal{O}\Big(\frac{T'(n)}{\delta'(n)}\Big).\end{aligned}$$

□

2.3 Constructing PRNG

There are multiple ways to construct a new PRNG. We have mentioned earlier in the text that we can use reductions to transform secure primitives. For example, it is possible to reduce a one-way function into PRNG. Another straightforward technique to create a new PRNG is to expand an existing. Both these techniques are reviewed below.

2.3.1 Expansion

As we have already mentioned above, CSPRNG produces a sequence of pseudo-random numbers from a selected input seed. Also, every generator must fulfill properties of Next-bit unpredictability and Undistinguishability from truly random sequences.

It is not easy to construct a function that would satisfy stated requirements. For that reason we might need to expand existing function that is already proved to be secure: we would like it to generate as wide and big output as possible without loss in security.

Another problem with using a pseudo-random number generator is that size of its output may not be sufficient for a given application. The available pseudo-random number generator might be capable of expanding by just one bit, while in many applications it is critical that the length of its output be significantly larger than its input. So it is important to be able to receive the pseudo-random number generator bits online. The following concept demonstrates that it is possible to construct a pseudo-random generator that can expand by an arbitrary polynomial amount of bits using a pseudo-random number generator that can only expand by one bit.

Stretching construction: Let $g : \{0,1\}^n \longrightarrow \{0,1\}^{n+1}$ be a pseudorandom number generator. Let $g^0(x)$ be equal to x, and $g^1(x) = g(x)$. For all $i > 1$ define:

$$g^i(x) = (g(x)_1, g^{i-1}(g(x)_{2,\dots,n+1}))$$

Based on [2] such a construction allows to stretch g such that $g^{l(n)}$ is also a PRNG, and reduction is linear preserving as proved in the source [2].

2.3.2 Extension from one-way permutation

As the chapter name proposes, we can design reduction from one-way functions to create a PRNG. We already know that PRNGs exist if and only if one-way functions exist. Before diving into construction of a PRNG from a general one-way function, let us firstly show a construction from it's less general instance, one-way permutation. It has a very favorable property of being bijective.

Construction and its proof rely on concepts of Inner Product Bit and Hidden Bit Theorem. Combining theoretical results with a stretching construction, we will be able to construct a PRNG, that stretches n input bits to $l(n)$ output bits.

In the [2], an inner product bit is defined as follows:

Definition 24 (Inner Product Bit)**.** [2, p.64]

Let $f \in \{f_n : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}\}$ be an instance of a family of polynomial-time functions, where the input is private and thus the security parameter is n.

Let $z \in \{0,1\}^n$. Define the inner product bit of f with respect to $f(x)$ and z to be $x \odot z$. Let $x, z \in \{0,1\}^n$ be chosen uniformly at random. Let A be an adversary. The success probability (prediction probability) of A for the inner product bit of f is

$$\delta(n) = |\Pr[A(f(x), z) = x \odot z] - \Pr[A(f(x), z) \neq x \odot z]|.$$

Then the inner product bit of f is $S(n)$-secure hidden if every adversary has a time-success ratio at least $S(n)$.

In other words, the inner product bit of f is hidden if no adversary can effectively retrieve it knowing only publicly available parameter z and output $f(x)$.

Theorem 3 (Hidden Bit Theorem). *[2, p.65] If f is a one-way function then the inner product bit of f is hidden. The reduction is poly-preserving.*

The proof is stated in the source [2].

The corollary of the Hidden Bit theorem states that for every one-way function f, its arbitrary input x and any random string z of the same size as x, and random bit b sequences $(f(x), x \odot z, z)$ and $(f(x), b, z)$ are computationally indistinguishable.

This corollary allows us to construct a following PRNG from a one-way permutation [2, p.74].

Construction of a PRNG from a one-way permutation

Let $l(n) = n^{\mathcal{O}(1)} > n$. Let $f : \{0,1\}^n \longrightarrow \{0,1\}^n$ be a one-way permutation. Define polynomial-time function ensemble $g : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$:

$$g(x, z) = (x \odot z, f(x) \odot z, f^2(x) \odot z, \dots, f^{l(n)}(x) \odot z),$$

where f^i is the function f composed with itself i times. z is a public input.

Theorem 4. *Function g is a PRNG.*

Proof. We need to use the stretching construction mentioned above to reach desired result. Namely, the following construction:

$$g^i(x) = (g(x)_1, g^{i-1}(g(x)_{2,\dots,n+1})).$$

From [2, p.52] we know that if g is PRNG then $g^{l(n)}$ is also a PRNG, and reduction is linear preserving.

So, now we want to prove, that $(x \odot z, f(x) \odot z, f^2(x) \odot z, \dots, f^{l(n)}(x) \odot z)$ is PRNG given that f is one-way permutation.

Define $g = (x \odot z, f(x))$. Then it follows from extension construction that

$$\begin{aligned}
g^0(x) &= x \\
g^1(x) &= (g(x)_1, g^0(g(x)_{2,\dots,n+1})) \\
&= (x \odot z, g^0(f(x))) = (x \odot z, f(x)) \\
g^2(x) &= (g(x)_1, g^1(g(x)_{2,\dots,n+1})) \\
&= (x \odot z, g^1(f(x))) = (x \odot z, (f(x) \odot z, f(f(x)))) \\
g^3(x) &= (g(x)_1, g^2(g(x)_{2,\dots,n+1})) \\
&= (x \odot z, g^2(f(x))) = (x \odot z, (f(x) \odot z, f(f(x)) \odot z, f(f(f(x))))))
\end{aligned}$$

To make notation more straightforward and tidy, denote function f composed with itself i times as f^i:

$$g^3(x) = (x \odot z, f(x) \odot z, f^2(x) \odot z, f^3(x)).$$

If we extend this construction repeatedly $l(n)+1$ times, we obtain the following result:

$$g^{l(n)+1}(x) = (x \odot z, f(x) \odot z, f^2(x) \odot z, \ldots, f^{l(n)}(x) \odot z, f^{l(n)+1}(x)).$$

Hence, $g^{l(n)+1}$ is a PRNG constructed from g. Limiting the function to hide all outputs of f, let the final public output of the construct be restricted to only $(x \odot z, f(x) \odot z, f^2(x) \odot z, \ldots, f^{l(n)}(x) \odot z, f^{l(n)+1}(x))$, while the last part, $f^{l(n)+1}(x)$, can be omitted on output.

For any $x \in \{0,1\}^n$, $f(x)$ uniquely determines x as $f(x)$ is a permutation. So no entropy is lost by the application of f and hence as f is uniformly distributed, $f(x)$ already looks random.

□

2.3.3 Extension from one-way function

Construct a PRNG from a regular one-way function (instead of permutation) requires certain changes and additional tools. That is because in case of regular functions we need to count with a difference of size of a preimage and range of the function, we need to consider stochastic distribution of input and output as distributions might be different for preimage and range.

The construction that incorporates all such factors and allows to create a pseudorandom generator from any one-way function requires defining following concepts:

- an entropy to measure amount of information,
- a family of $\sigma(n)$-regular one-way functions,
- degeneracy of function to measure an aggregated average number of distinct inputs mapped to the same output,
- a universal hash function to smooth the entropy using hashing.

Definition 25 (Entropy and information)**.** [2, p.79]

Let $\mathcal{D}_n$ be a distribution on $\{0,1\}^n$ and let X be chosen randomly with respect to $\mathcal{D}_n$. For every $x \in \{0,1\}^n$ the *information* of x with respect to X is defined as

$$\text{infrom}_X(x) = -\log\Big(\Pr[X = x]\Big),$$

the *entropy* of X is defined as the expected information of X:

$$\text{ent}(X) = E[\text{infrom}_X(X)].$$

Definition 26 (Range and preimages of a function)**.** [2, p.91]
Let $f : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ be a function. Define *range* as follows:

$$H_f(n) = \{f(x) : x \in \{0,1\}^n\}.$$

For every possible function value $y \in H_f(n)$ define *preimage* as follows:

$$\text{pre}_f(y) = \{x \in \{0,1\}^n : f(x) = y\}.$$

The intuitive notion of one-to-one function would correspond to the scenario, when all preimages are of size exactly 1. Some functions may map multiple input values to a single output value, resulting in a phenomenon known as degeneracy.

Definition 27 (degeneracy)**.** [2, p.91]
Let $\mathcal{D}_n$ be a probability ensemble with output length n. Let x be chosen in random from $\mathcal{D}_n$. Define the *degeneracy* of f with respect to $\mathcal{D}_n$ as

$$\text{degen}_f^{\mathcal{D}_n}(n) = E\Big[\text{infor}_x(X) - \text{infor}_{f(x)}(f(x))\Big].$$

Clearly, it is equivalent to $\text{degen}_f^{\mathcal{D}_n}(n) = \text{ent}(x) - \text{ent}(f(x))$.

Definition 28 (regular function)**.** [2, p.91]
Function $f : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ is $\sigma(n)$-regular if $|\text{pre}_f(y)| = \sigma(n)$ for all $y \in H_f(n)$.

It is not difficult to show that $\text{degen}_f(n) = \log(\sigma(n))$.
The last prerequisite is a universal hash function:

Definition 29 (universal hash function)**.** [2, p.84]
Let $h : \{0,1\}^{l(n)} \times \{0,1\}^n \to \{0,1\}^{m(n)}$ be a **P**-time function ensemble. Let Y be uniformly chosen from $\{0,1\}^{l(n)}$. We say h is a (pairwise independent) universal hash function if, for all $x, x' \in \{0,1\}^n$, $x \neq x'$, for all $a, a' \in \{0,1\}^{m(n)}$,

$$\Pr\left[h_Y(x) = a \ \wedge h_Y(x') = a'\right] = \frac{1}{2^{2m(n)}},$$

i.e. h_Y maps every distinct pair x and x' independently and uniformly.

For an easier notation for fixed $y \in \{0,1\}^{l(n)}$, $h(y,x)$ is viewed as a function $h_y(x)$ of x that maps (or hashes) n bits to $m(n)$ bits.

Poly-preserving construction for regular one-way functions. Let $f : \{0,1\}^n \to \{0,1\}^{l(n)}$ be a $\sigma(n)$-regular one-way function. Denote with $d(n)$ its degeneracy:

$$d(n) = \lceil \text{degen}_f(n) \rceil.$$

Let $r(n)$ be a positive integer valued function. Let $z \in \{0,1\}^{x\times(2r(n)+1)}$. Let h' and h'' be universal hash functions:

$$\begin{aligned} h' \ &: \{0,1\}^{l'\ (n)} \times \{0,1\}^{l(n)} \to \{0,1\}^{n-d(n)-r(n)}, \\ h'' &: \{0,1\}^{l''(n)} \times \{0,1\}^{n} \quad \to \ \{0,1\}^{d(n)-r(n)}. \end{aligned}$$

Under above assumptions define a pseudorandom generator

$$g(x, z, y', y'') = (h'_{y'}(f(x)),\ h''_{y''}(x), x \odot z, y', y'').$$

Omitting the formality of notation (y' and y'' are public inputs, to hash function, for function defined by some standard those would be a publicly known constant parameters), the simplified formula can be written as

$$g(x, z) = (h'(f(x)),\ h''(x), x \odot z).$$

Here, we simplify the motivation behind this construction. A hash function is defined to have a property to uniformly distribute its outputs, so even if x or $f(x)$ are distributed in a special way, the hash function 'normalizes' them. That is why hash functions are essential in construction. Next, only a part of $f(x)$ is used along with x as input for hash functions. That is due to an entropy, an "amount of information" in those. Parameter $d(n)$ serves to measure the amount of entropy lost in the function f and h' extracts just the limited volume of data, while $h''(x)$ returns it into the game directly from x, omitting intermediate function f. Only the parameter $r(n)$ is not explained yet. It is needed simply to ensure that security is preserved. The reduction of f to g is poly-preserving when $r(n)$ is set to the logarithm of the security of f.

Detailed proof that reduction of f to g is poly-preserving and g is PRNG is stated in the source, including proofs of intermediate lemmas.[2, p.93]

3. Cryptographically secure PRNG used in practice

The need to use cryptographically secure random number generators (CSPRNG) is due to their ability to provide a high degree of unpredictability, which is critical for security in areas such as data encryption, key generation and authentication. The practical alternative of pseudo-random sequences generated by deterministic algorithms are truly random sequences collected from sources that are considered unpredictable. The most common solution of the need combines both options: "expensive" and limited in amount True Randomness is collected to later be used to initialize the internal state of PRNG that can produce "cheaper" randomness. True Randomness is expensive, because it almost always requires specialised hardware or communication with a third party who collects entropy. While PRNG is just an algorithm that usually can run on the same device.

Even though, the previous chapters demonstrate that it is not known if they exist (existence is conditioned by assumption that $\mathbf{P} \neq \mathbf{NP}$), modern communication technologies are in dire need of using algorithms that are as close as possible to theoretically ideal.

In this chapter, we will review some examples of such generators. From now on, this text refers to the notion such as generator or CSPRNG as to a practical instance of algorithms.

The list of most used generators includes:

- **Fortuna**: the generator is based on running a block cipher in the counter mode of operation. By design, the block cipher is not chosen and any trustful and secure primitive can be used. Practical Cryptography[15, p.161] suggests AES, Serpent or Twofish. Fortuna also includes the entropy accumulator that collects and stores randomness from various sources to introduce new seed to restart the generator before periodicity shows itself [23].

- **Yarrow**: similar as it's successor, Fortuna mentioned above, Yarrow generator uses a block cipher in counter mode to generate pseudo-random bytes. The main reason for deprecation of Yarrow was the fact that it is tightly tied to the DES cipher, used as a block cipher, and SHA-1 hash algorithm applied for a reseeding [22]. Both these primitives are not trustworthy today, because of multiple attacks developed lately. DES is not recommended since standard withdrawal in 2005 [18]. SHA-1 is not recommended by NIST since 2011, when SHA-2 became the standard [17].

- **ChaCha20**: the generator is used for the arc4random generating function in the OpenBSD (a security-focused, free and open-source, Unix-like operating system based on the Berkeley Software Distribution - BSD). ChaCha20 is based on the stream cipher encryption algorithm, it belongs to the Salsa family of ciphers. It uses 32-bit operations and 10 iterations of the double round, a high degree of cryptographic strength and exceptional performance [27]. ChaCha20 replaced RC4 and DragonFly BSD which both are not considered trustful anymore.

- **Blum Blum Shub**: proposed in 1986 and proved to be at least as difficult to solve as the quadratic residuosity problem (hence to be trustful). It consists of repeatedly performing the operation of squaring modulo M, where M is the product of two sufficiently large prime numbers. [28].

- **ISAAC** (Indirection, Shift, Accumulate, Add, and Count): proposed by Robert Jenkins [29, section 6.3]. Similarly to RC4, it operates by utilizing a 256 array of 32-bit integers as its internal state, with results written to a separate array and read sequentially until exhausted. This process involves a series of computations with array elements, an accumulator, and a counter. Generation takes just 19 32-bit operations to produce an output string of the same length, making ISAAC highly efficient on 32-bit platforms.

  ISAAC author has also published a version for 64-bit platforms on his web.

In the following text we will take a closer look at the mentioned algorithms. These generators have different properties and are used in different areas, depending on the requirements for safety and performance. As an example of a Cryptographically Not Secure Pseudorandom Numbers Generator, we will take a closer look at a Mersenne Twister - a generator which, among other things, is used in the default mechanism for obtaining random numbers in the popular Python language.

3.1 Yarrow

Yarrow is an open-source algorithm offered by Schneier et al. [22]. The most known variation of Yarrow-160 is designed to be used with 3-DES as a symmetric cipher and SHA-1 as a hash function. But generally, it can use any symmetric cipher and any hash function.

In the paper authors defined Yarrow algorithm to consist of the following major components:

- Entropy Accumulator
- Reseed mechanism
- Generation mechanism
- Reseed control

Entropy accumulation is the process used to set a new unpredictable internal state of PRNG. As any generation of pseudorandom sequence is a deterministic algorithm, the important security prerequisite is unpredictable input. To enable such unpredictable reseed, external entropy has to be collected. Entropy accumulator is a way to store these random inputs. According to Yarrow's design, they are collected into two pools just concatenating new inputs to those existing. Once enough entropy is collected, a hash function is applied to the entire content of the accumulator.

The reseed mechanism is a protocol to change the internal state of the generator.

For arbitrarily chosen hyperparameter t reseed mechanism is designed in the following way:

Algorithm 3 Reseed mechanism

Inputs: `entropy_accumulator` (entire content of the entropy accumulator)
Prerequisites and notation: $h : \{0,1\}^* \to \{0,1\}^c$ is a hash function with output of length c.
E_K is the generator symmetric cipher.
Outputs: K - new generator key, C - generator counter, seed file.

```
1: $v_0 \leftarrow h(\text{entropy\_accumulator})$
2: for $i = 1, \ldots, t$ do:
3:     $v_i \leftarrow h(v_{i-1}|v_0|i)$
4: end for
5: $K \leftarrow h(h(v_t, K), k)$                          ▷ new generator key
6: entropy_accumulator $\leftarrow 0$          ▷ Reset the entropy accumulator.
7: Wipe all used interstate variables.
8: $C \leftarrow E_K(0)$                           ▷ Set generator counter.
9: if the seed file is in use then
10:     Rewrite the seed file with the first $2k$ bits of generator output.
11: end if
```

The generator. Generator is a functional part of the system.

It uses any symmetric cipher E_K in a counter mode, where key K and counter C serves as an internal state of the generator.

Algorithm 4 Yarrow Generator

Input: C - counter of the generator,
K - the hidden key.
Prerequisites and notation: E_K - chosen for the generator symmetric cipher.
Output: R - pseudorandom bits.

```
1: $C \leftarrow C + 1 \mod 2^n$
2: $R \leftarrow E_K(C)$
3: if System time-frequency security parameter exceeded then
4:     $K \leftarrow$ next $k$ bits of the output.          ▷ Apply generator gate.
5: end if
```

R is the next output block of the generator.

Note that the "generator gate" (steps 3-5 of the algorithm above) is an improvement of the algorithm. Security parameter can be set based on system's requirements. In Yarrow-160, authors offer to use three-key triple-DES in counter mode to generate outputs, and to apply the generator gate every ten outputs.

Generally, it is not the rule. And it is possible to set similar algorithm with different cipher running in a different mode.

3.2 Fortuna

Fortuna is an environment built over the Yarrow algorithm. It was designed by Bruce Schneier and Niels Ferguson in 2003 [23], - 4 years after they published Yarrow mentioned in the text above.

The generation algorithm itself was not changed. It uses the same principles as the Yarrow Generator: continuously running the block cipher in counter mode, encrypting the counter C in every round. What is different is the block size, key and cipher. Instead of triple-DES in the case of Yarrow, Fortuna recommends applying ”an AES-like block cipher for the generator; feel free to choose AES (Rijndael), Serpent, or Twofish for this function.” [23]. Hence, the generator's internal state consists of a 256-bit key and a 128-bit counter (for comparison previous version of Yarrow-160 used 168-effective-bit key (in every byte of each of the three 64-bit keys the last bit is a parity bit; it is a peculiarity of DES). Generation changed the primitive function, but patterns stayed the same. Much more significant change is observed in entropy accumulation. Instead of 2 pools, now 32 pools are introduced. The overall design is generally very similar to that of its predecessor. Some parameters have been increased to enhance security; more secure cryptographic primitives are used.

Fortuna generator is a popular solution in practise. For example, Apple implements it in the iOS, iPadOS, macOS, tvOS and watchOS kernels. [24]

3.3 ChaCha20

ChaCha is a family of cryptographic functions introduced by Daniel J. Bernstein in 2008 [27]. It is closely related to the Salsa family designed by Berstein in 2005. Both are stream ciphers that are built up on an add-rotate-XOR pseudorandom function. ChaCha's internal state consists of 512 bits ordered as matrix 4x4 of 32-bit words. The initial state is set from the combination of 256-bit key, 128-bit constant, 64-bit counter and 64-bit nonce. The constant is ”expand 32-byte k” encoded with ASCII (hexadecimal): 0x65787061 0x6e642033 0x322d6279 0x7465206b.

A quarter round (QR(a, b, c, d)) is a fundamental operation in ChaCha that performs a series of operations on four words of the internal state. It is called a ”quarter round” because it operates on a quarter of the state array at a time. QR(a, b, c, d) performs the following transformations and returns words a, b, c, d, where “$word << i$” denotes bits left shift by i positions:

$$\begin{aligned}
&a \mathrel{+}= b; \quad d \oplus a; \quad d << 16;\\
&c \mathrel{+}= d; \quad b \oplus c; \quad b << 12;\\
&a \mathrel{+}= b; \quad d \oplus a; \quad d << 8;\\
&c \mathrel{+}= d; \quad b \oplus c; \quad b << 7;
\end{aligned}$$

3.4 ISAAC

The generator was offered in 1996 together with it's 2 predecessors IA and ABAA [35]. Requirements for every next version of the generator were adjusted based on performance of the previous.

ISAAC always produces 256 bytes of pseudo-random output. The generator operates on 32-bit words (the design was offered in 1996, when 64-bit infrastructures have not been too popular yet), its state assembly from the following elements. Following the example from the author of cryptoanalytical attack [36], we will index every variable to track the state over time. E.g. mm_t is the state of variable mm at time t.

Alias	**Role (Explanation of its function)**	**Size (Bytes)**
mm_t	Internal state array used to generate random numbers. It is modified during each iteration of the 'isaac' function.	1024
aa_t	Accumulator; it is an internal state variable that gets modified by bitwise operations and affects the generation of random numbers.	4
bb_t	Previous result; another internal state variable that contributes to the generation of random numbers and is updated in each iteration.	4
cc_t	Counter; an internal state variable that increments once per 256 results and affects the generation of random numbers.	4
randrsl_t	Results array where the output sequence of random numbers is stored.	1024

Table 3.1: Variables used in the ISAAC random number generator

To simplify the notation in the text below, let us introduce the following function (it replicates function $G()$ in notation from [36] ($word << i$ denotes bits left shift by i positions:):

$$\text{Shift}(\text{aa}_{t-1}, i) = \begin{cases} \text{aa}_{t-1} \oplus (\text{aa}_{t-1} << 13); & \text{if } i \pmod 4 = 0 \\ \text{aa}_{t-1} \oplus (\text{aa}_{t-1} >> 6); & \text{if } i \pmod 4 = 1 \\ \text{aa}_{t-1} \oplus (\text{aa}_{t-1} << 2); & \text{if } i \pmod 4 = 2 \\ \text{aa}_{t-1} \oplus (\text{aa}_{t-1} >> 16); & \text{if } i \pmod 4 = 3 \end{cases}$$

With the notation for the ISAAC algorithm established, we can now proceed to detail the steps of the algorithm itself, as shown in Algorithm 5.

Algorithm 5 ISAAC [35, 36]

```
Inputs: $t, \mathrm{aa}_{t-1}, \mathrm{bb}_{t-1}, \mathrm{cc}_{t-1}, \mathrm{mm}_{t-1}$
Prerequisites and notation: $\mathrm{Shift}(\mathrm{aa}_{t-1}, i)$
Outputs: $\mathrm{randrsl}_t$
1: $\mathrm{cc}_t \leftarrow \mathrm{cc}_{t-1} + 1$
2: $\mathrm{bb}_t \leftarrow \mathrm{bb}_{t-1} + \mathrm{cc}_t$
3: for $i = 0, \dots, 255$ do:
4:     $\mathrm{aa}_t \leftarrow \mathrm{Shift}(\mathrm{aa}_{t-1}, i) + \mathrm{mm}_{t-1}[i + 128 \pmod{256}] \pmod{2^{32}}$
5:     $\mathrm{mm}_t[i] \leftarrow \mathrm{mm}_{t-1}[(\mathrm{mm}_{t-1}[i] >> 2)] + \mathrm{aa}_{t-1} + \mathrm{bb}_{t-1} \pmod{2^{32}}$
6:     $\mathrm{bb}_t \leftarrow \mathrm{mm}_t[\mathrm{mm}_t[i] >> 10] + \mathrm{mm}_{t-1}[i] \pmod{2^{32}}$
7:     $\mathrm{randrsl}_t[i] \leftarrow \mathrm{bb}_t$
8: end for
```

Because the generator was not released in an analytical form, but only as a programming code, its notation seems very confusing compared to other PRNGs described above.

On the other hand, the security of ISAAC does not rely on any other cryptographical function, as it does not employ any secure cryptographical primitive, but plays such role itself. Such independence can be a key factor to prefer ISAAC over other options.

3.5 The ANSI X9.17 PRNG

This and following algorithms are discussed in [21] and mentioned several times in the next section. We introduce just a brief description without much details, because these PRNGs were almost outdated at the time of the release of [21]. Today some of them might be hacked within minutes.

The ANSI X9.17 is a standard developed by the Accredited Standards Committee on Financial Services, X9, operating under the procedures of the American National Standards Institute [39]. Standard recommends the way keys should be created, stored and used, describing surrounding infrastructure like key entry standards and Cryptographic service messages structure and format. The standard requires that ”keys shall be generated so that keys and IVs are random or pseudorandom” and an example of PRNG is described in the appendix C of [39]. Although, it is not required to use given PRNG, it is the only such algorithm mentioned in the standard. PRNG requires a use of Data Encryption Algorithm (DEA) as describe in the ANSI X3.92 1981 standard [40]. This cipher is usually referred as DES.

Initialization. The generator requires two random vectors to be generated prior to its first use: a secret DES key K, and a seed value V. K is not reset at any time. V is changed every cycle.

Generation. To generate i-th random sequence, firstly, a temporary value T_i is computed using current timestamp:

$$T_i = E_K(\text{timestamp}).$$

Generated random bytes are obtained in the following way:

$$\text{output}[i] = E_K(T_i \oplus V_i).$$

And finally, the seed should be updated:

$$V_i = E_K(T_i + \text{output}[i]).$$

3.6 The DSA PRNG

Generator intended for generating pseudorandom parameters for the DSA signature algorithm relies on the hash function primitive - SHA algorithm. Due to the fact that the generator was approved by NSA[37], it was used for other purposes as well as for Digital Signature Standard. Digital Signature standard in its original statement from 1994 [38] orders to use PRNG described below or "other FIPS approved security methods". It is supplemented with a reference to above ANSI X9.17 PRNG: "One FIPS approved pseudorandom integer generator is supplied in Appendix C of ANSI X9.17,"Financial Institution Key Management (Wholesale)." [39]

The generator operates on states X_i. Every output is produced as a hash of current state combined with an optional parameter W_i. The next state X_{i+1} is set using a combination of current state X_i, and negation of its output $1 \oplus \text{output}_i$.

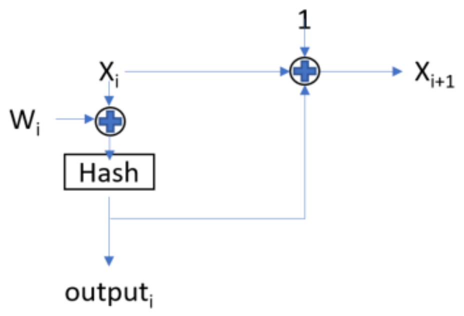


Figure 3.1: DSA scheme

In later chapters we will refer to this PRNG when studying attacks offered by Schneier [21]

3.7 The RSAREF PRNG

RSAREF is the free library RSA Data Security, Inc. made available for the purpose of implementing freeware applications. It includes implementation of several encryption algorithms, including RSA.

RSAREF is an open source software. It is available, for example on the Finnish University and Research Network [20]. Code is written in C. File *r_random.c* can be used separately from others (conditioned by importing dependencies as md5 module). So, technically, it is an independent random numbers generator. Following rather the notation from the source [21] then from the code itself the RSAREF PRNG includes following elements:

- 16 bytes counter C_i - the state
- method *R_RandomUpdate* to update the state with optional input X:

$$C_{i+1} = C_i + \mathrm{MD5}(X) \bmod 2^{128}$$

- method *R_GenerateBytes* to produce output and update the state:

$$\begin{aligned}\mathrm{output}_i &= \mathrm{MD5}(C_i) \bmod 2^{128}\\ C_{i+1} &= C_i + 1 \bmod 2^{128}\end{aligned}$$

Additional detail visible in the source code is that pseudorandom numbers are generated by method *R_GenerateBytes* in the requested quantity. And if requested quantity is not a multiple of 16, some bytes would stay precomputed and returned next time a randomness is requested. This detail is important, but not critical. It only implies that not always current output depends on latest state. Part of it might have been created using the previous state.

4. Practical attacks on PRNGs

In the next part of the thesis, we will look at different ways that security of random number generators have been broken. We will focus on four real-world examples:

The MIFARE Crypto-1 case, where the security of a common smart card was defeated. The DUAL_EC_DRBG incident, where a number generator used for security had a secret weakness. The PS 3 attack, showing that even popular game consoles can be hacked due to significant security design flaws. The Mersenne Twister, which is a number generator that is not safe for keeping things secret. These stories will help us understand the various challenges in keeping our digital world secure.

4.1 MIFARE Crypto-1

MIFARE is a brand focused on the production of a series of integrated circuit chips of various levels of the ISO/IEC 14443 Type-A 13.56 MHz contactless smart card standard. MIFARE offer different products with different level of security and computational requirements. Here we want to focus on one exact family of their products: MIFARE Classic. It uses a security protocol Crypto-1. Cards with mentioned solution were widely used in different systems: to manage access, to verify the owner, as a small payment system. For example, these chips were used in Czech public transport: Hradecka Karta and Pardubicka Karta. Prague public transport relies on more modern chips MIFARE DESFire, which was introduced shortly after the first attacks on the MIFARE Classic family. Chips are used in multiple countries on every continent.

The first vulnerability was discovered in 2007 and at that time attack offered was not too efficient, but still according to [34] could be done in a matter of hours rather than years for a brute force attack. This weakness is hidden in the Random Number Generator required for the Crypto-1. As such, the schema includes a LFSR generated by polynomial $x^{16} + x^{14} + x^{13} + x^{11} + 1$. Even though the register has 32 bits, it works with only 16 of them. Hence, a cycle is closed after 2^{16} iterations.

In a parallel investigation, [31] delves into the weaknesses of the pseudorandom generator on the MIFARE Classic card. The analysis exposes a practical attack leveraging the weakness of the generator to facilitate a known plaintext brute-force approach. As the first step in their attack, the authentication step should be overcome.

By repeatedly requesting card nonces, the study reveals the same conclusion as [34] without deep analysis of the chip's hardware. The nonce, generated by a Linear Feedback Shift Register (LFSR), exhibits predictability, with the potential to reappear after 0.618s. However, this time is feasible only in theory. In practice, nonce reappears only a few times per hour - simply because not every output of LSFR is used, but only those requested at the moment when nonce is needed.

Another weakness of the PRNG used in MIFARE was discovered in "Reverse-Engineering a Cryptographic RFID Tag" [34] and confirmed in "A Practical Attack on the MIFARE Classic" [31]. The generating LFSR is reset to a known state every time the tag starts operating. This leads to the fact that the response to the nonce request sent in a fixed time after powering-up the card is very likely to be the same. Responses might differ due to a timing inaccuracy, but can still be reduced to just 10 different values of nonce.

Conclusion: however a single weakness of PRNG does not immediately break apart entire security, the revelation that a skilled attacker can initiate authentication with precisely the same nonce underlines the urgency for addressing these inherent weaknesses in the MIFARE Classic protocol to ensure robust security in practical deployment scenarios.

4.2 DUAL EC DRBG

DUAL_EC_DRBG is a widely discussed algorithm for generating random numbers, encouraged by the NSA in the early 2000s. There is not much information publicly available about when the development of the algorithm began and ended. However, it is believed that the NSA has deliberately and systematically pursued a policy to introduce vulnerabilities into Internet security products in a variety of ways. However, these opinions are mainly based on statements from industry participants and former employees. There are no instructions for use and no direct evidence that there is a "back door" specifically and deliberately designed for inclusion in the standard. Only a hypothesis that existence of such is possible.

Description of the PRNG appeared firstly in ANSI X9.82 draft and then in an approved standard. It was also included into ISO/IEC 18031:2005. Implementations were included into multiple cryptographic libraries: RSA Security (BSAFE library), OpenSSL, Microsoft, Cisco and further. But only BSAFE used it by default.

Design. The generator includes:

- a state s_i,
- a function $g_P(s_{k-1})$ used to produce new state s_k every cycle,
- and a function $g_Q(s_k)$ to generate random sequence r_k every cycle.

The state s must be reseeded once in $reseed\_interval$ (it is given by the standard), values of P and Q are suggested by the standard constants - points on the elliptic curve P-256 (or P-384 or P-521). Algorithm operates over finite field $F_p(\mathbb{Z}/p\mathbb{Z})$, p is determined by the standard. And the curve is given by equation

$$y^2 = x^3 - 3x + b,$$

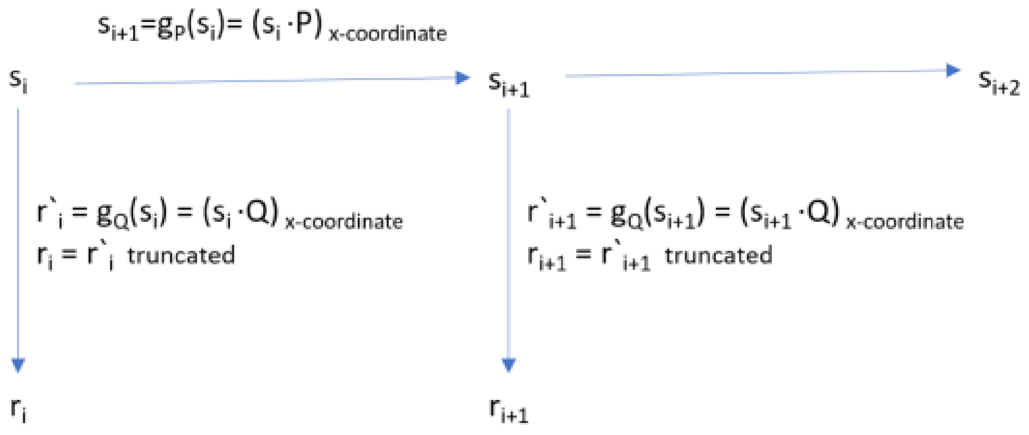
where b is determined by the standard.

Function X used below is defined to extract x-coordinate: $X(x, y) = x$.

Functions $g_P(x)$ and $g_Q(x)$ are defined as follows:

$$g_P(x) = X(x \cdot P),$$
$$g_Q(x) = X(x \cdot Q).$$

The output is a truncated x-coordinate of the resulting point. Depending on the setup $g_Q(x)$ is truncated to rightmost 240, 368 or 504 bits (for P-256, P-384 or P-521 corresponding).



s_{i+1}=g_P(s_i)= (s_i ·P) x-coordinate
s_i
s_{i+1}
s_{i+2}
r`_i = g_Q(s_i) = (s_i ·Q) x-coordinate
r_i = r`_i truncated
r`_{i+1} = g_Q(s_{i+1}) = (s_{i+1} ·Q) x-coordinate
r_{i+1} = r`_{i+1} truncated
r_i
r_{i+1}

Figure 4.1: DUAL_EC_DRBG scheme

DUAL_EC_DRBG is based on the elliptic curve discrete logarithm problem: given points A and B on an elliptic curve, find x such that $A = x \cdot B$. However, the discrete logarithm problem might be much easier for the one who knows certain relation between points P and Q given in standard. Specially, in case that P and Q are defined to fulfill $P = z \cdot Q$ and arbitrary z is known by the organisation that offered constant values for the standard. This is one of hypothesises for the 'backdoor' to exists. [32]

Following scheme above, the intruder, who knows the output r_i, can brute force truncated 16 bit to guess $s_i \cdot Q$. Assume that intruder posses z such that $P = z \cdot Q$. Then, he might compute:

$$\begin{aligned} & z \cdot s_i \cdot Q \\ &= s_i \cdot z \cdot Q \\ &= s_i \cdot P \\ &= s_{i+1} \end{aligned}$$

Having guessed correctly $s_i \cdot P$, intruder obtains a future state of PRNG and posses ability to replicate future outputs of PRNG until first reseed.

The value of such z value has not been published yet, and it has not been proven that someone has ever computed it. But insiders reported media, that NSA made a lot of efforts to ensure that the scheme had a back door and that the government could use it at its discretion.

4.3 PS 3

The basic idea of the weakness here does not refer to an insecure generator, but to a missing generator. According Fail0verflow presentation at the 2010 Chaos Communication Congress, the group forged the software signature only because the random number was not random per se. Details were not publicised, but one can deduce from their presentation that the predefined constant was used as a nonce for ECDSA.

4.4 Mersenne twister

The Mersenne Twister is not suitable for cryptographic purposes where high-security standards are required as it is not cryptographically secure.

The PRNG has a very long period of the chosen Mersenne prime. The most used version MT19937 has a period of $2^{19937} - 1$. The algorithm below uses constants for MT19937. Mersenne twister has an internal state of 624 32-bit words array. It is seeded in a sophisticated way, but the generation algorithm itself is simple:

Algorithm 6 GetRandom(MT - array with internal state, index - current position in MT)

```
if index ≥ 624 then
    MT = twist(MT)
    index = 1
end if
$y \leftarrow$ MT[index]
$y \leftarrow y \oplus ((y >> 11) \wedge$ 0xffffffff)
$y \leftarrow y \oplus ((y << 7) \wedge$ 0x9d2c5680U)
$y \leftarrow y \oplus ((y << 15) \wedge$ 0xefc60000U)
$y \leftarrow y \oplus (y >> 18)$
```

Function twist() rekeys internal state. Same as the generation function, it uses only add-rotate-XOR operations and ordinary constants.

Attack. From the definition of the function it is easy to see, that one output y_i depends only on 1 word from internal state $MT[i]$. When 624 outputs in sequence are known, one can try to find corresponding internal states. To do so SAT solver can be used. Idea an attack is inspired by an article [33]. Compared with the original idea, we have managed to predict randomness generated by a built-in module of Python programming language.

Here is an example of such an attack in Python:

```
def untemper(out):
    y1 = z3.BitVec('y1', 32)
    y2 = z3.BitVec('y2', 32)
    y3 = z3.BitVec('y3', 32)
    y4 = z3.BitVec('y4', 32)
    y = z3.BitVecVal(out, 32)
    s = z3.Solver()
    equations = [
        y2 == y1 ^ (z3.LShR(y1, 11)),
        y3 == y2 ^ ((y2 << 7) & 0x9D2C5680),
        y4 == y3 ^ ((y3 << 15) & 0xEFC60000),
        y == y4 ^ (z3.LShR(y4, 18))
    ]
    s.add(equations)
    s.check()
    return s.model()[y1].as_long()
```

Z3 is a high-performance theorem prover developed at Microsoft Research, here used in Python.

Recovering internal state from outputs requires all outputs between 2 sequential calls of twist(). After the internal state is known to an attacker, he can set up his own copy of MT and proceed with the generation of the same sequence. The entire attack takes about 10 seconds on a PC with a processor Intel(R) Core(TM) i5-8350U. The code is available in an appendix A.

Overall neither Mersenne Twister nor the attack is complicated from cryptographic point of view. This section serves to show the importance of using CSPRNG. Mersenne Twister is known and widely used for its good statistical properties and distribution, while security is not a goal. MT19937 is used as a core PRNG in a popular nowaday language Python. Library 'random' seeds MT19937 with a system time or with user-defined seed and transforms obtained 32-bit words to correspond range and format required by user.

While practical attacks expose significant vulnerabilities in cryptographic systems, cryptanalytic attacks are often still too complex on conventional devices. They often reveal only the subtle nuances of cryptographic schemes, which leads to better understanding and subsequent strengthening of security measures, but not so often to an absolute loss of security in an instant.

5. Cryptoanalytical attacks on PRNGs

According to [21] attacks on a PRNG can be classified, and among classes of attacks, the authors identify the following as the most common:

- Direct Cryptoanalytic Attack.
- Input-Based Attack.

  Input attacks may be further divided into known-input, replayed-input, and chosen-input attacks.
- State Compromise Extension Attacks.
  - Backtracking Attacks.
  - Permanent Compromise Attacks.
  - Iterative Guessing Attacks.
  - Meet-in-the-Middle Attacks.

5.1 Cryptanalysis of the Random Number Generator of the Windows Operating System

In [19], the authors did research on the weaknesses of the Microsoft CryptoAPI (programming interface that allows developers to secure their Windows-based applications) utility CryptGenRandom that provides a cryptographically secure PRNG function. The report on the weaknesses was published in 2007, and according to public sources, the weaknesses were fixed by Microsoft by the mid of 2008. CryptGenRandom, or WRNG (Windows Random Number Generator) as it is referred to in the source, was used in Windows 2000 and Windows XP. Microsoft never released the source codes to public, so the study was done with only a functional version of Windows.

In the study, the first step was to perform an analysis of the Binary Code. It is worth noting that already in the first step of the study, the authors confirmed the correctness and importance of Kerckhoffs's principles. These were briefly stated by Stephen M. Bellovin as "design your system assuming that your opponents know it in detail". Even though Microsoft's code is not public, the cryptographic community knows their algorithms.

The authors translated functional implementation of the binary assembly into the pseudo-code. Having in hand a version of the code written in a high-level programming language the authors described the algorithm of the CryptGenRandom as follows.

Algorithm 7 CryptGenRandom(Buffer, Len)

```
1: Outputs Len bytes to Buffer
2: while Len > 0 do
3:     R := R ⊕ get_next_20_rc4_bytes()
4:     State := State ⊕ R
5:     T := SHA-1(State)
6:     Buffer := Buffer | T
7:     R[0..4] := T[0..4]
8:     State := State + R + 1
9:     Len := Len - 20
10: end while
```

All internal variables are 20 bytes long and uninitialized. In the context of assembly, an uninitialized variable might have any initial value depending on the stack content. But this does not guarantee that its value is truly random as the stack is located in a certain part of the memory dedicated to the certain software. Hence, values stored in the stack might be, and according to authors are, highly correlated: initial vectors are not truly random, and neither are they constant.

Every process has its instance of CryptGenRandom running. That means that should the user work in 2 distinct applications in parallel, each of them would have different independent states of Windows PRNG. On the other hand, only State and R are stored on the stack, while RC4 states and auxiliary variables are located in DLL (Dynamic Link Library). Overall, the security of different processes does not depend on other processes, one of which could be an attacker. But on the other hand, this scheme with only one consumer per instance leads to the low frequency of reinitializing the PRNG internal state.

Function `get_next_20_rc4_bytes()` holds 8 instances of a stream cipher RC4. On each consecutive call i it returns 20 bytes of $(i \mod 8)$-th stream. When $(i \mod 8)$-th stream returned more than 16 KB, its key is reset using Windows' internal source of entropy.

An attack assuming knowledge of `CryptGenRandom` state at time i

The following part as a whole assumes that an adversary somehow revealed the internal state of PRNG, namely variables State, R and state of all RC4 stream ciphers. At the first sight, these assumptions look impossible to happen, but as described by authors, under certain circumstances they might be disclosed almost instantly. According to the source, even though the value of the uninitialised variable could look relatively random, it did not seem so at the time experiment was conducted. New instances of `CryptGenRandom` are called again after the previous process was halted and instances created in parallel usually are highly correlated (Hamming distance of 10 for 160-bit words attests this statement).

The primitive is considered backward secure if knowledge of the current state does not reveal any information about future states or outputs. This is not the case for any deterministic algorithm: knowledge of the state and algorithm itself leads to an intuitive complex attack on backward security. The attacker needs to run a simulation of `CryptGenRandom` with chosen parameters.

The primitive is forward secure if, from knowledge of its i-th state, it is impossible to derive previous states or outputs. The following text describes authors' forward-security attack on `CryptGenRandom`.

The PRNG relies on RC4 for generating pseudo-random output streams, which are combined with the generator's state. RC4 is a strong stream cipher but lacks forward security. This means that if the current state of an RC4 cipher is known, it is possible to calculate its previous states and outputs with low complexity. Hence, the difficult part in getting all previous states of the generator is to get all previous values of variables State, R.

In the manner similar to the authors, we will denote:

- R^t - the state of R in the t-th loop of the main loop.
- S^t - state of State in the t-th loop of the main loop.
- $\mathsf{R}^{t,i}$ or $\mathsf{S}^{t,i}$ - a state in t-th loop before executing line i of the algorithm.
- RC^t denotes the output of get_next_20_rc4_bytes() in the t-th iteration.
- And X_L denotes the left-most 120 bits of X; X_R - right-most 40 bits of X.

An attack with an overhead of 2^{40}. The goal is to recover R^{t-1}, S^{t-1} from given R^t, S^t, RC^t. RC^{t-1} can be computed without any additional complications. It can not be considered as a one-way function when its internal state is known. Based on the code the following relations can be observed.

$$\begin{aligned}
\mathsf{S}^{t-1,11} &= \mathsf{S}^t - \mathsf{R}^t - 1 \\
\mathsf{R}^{t-1,\,9} &= \mathsf{R}^t_L|\{0,1\}^{40} \qquad\qquad \text{right-most 40 bits are not known at this moment} \\
\mathsf{R}^{t-1} &= \mathsf{R}^{t-1,9} \oplus \mathsf{RC}^{t-1} \\
\mathsf{S}^{t-1} &= \mathsf{S}^{t-1,5} = \mathsf{S}^{t-1,11} \oplus \mathsf{R}^{t-1,9} = (\mathsf{S}^t - \mathsf{R}^t - 1) \oplus (\mathsf{R}^t_L|\mathsf{R}^{t-1}_R)
\end{aligned}$$

The following relation is also observed from the code:

$$\mathsf{R}^t_R = \text{SHA-1}(\mathsf{S}^{t-1,11})_R = \text{SHA-1}(\mathsf{S}^t - \mathsf{R}^t - 1)_R$$

Applying it to $(t-1)$ states and substituting the above relations gives us

$$\mathsf{R}^{t-1}_R = \text{SHA-1}\Big(\big((\mathsf{S}^t - \mathsf{R}^t - 1) \oplus (\mathsf{R}^t_L|\mathsf{R}^{t-1}_R)\big) - (\mathsf{R}^t_L|\mathsf{R}^{t-1}_R) \oplus \mathsf{RC}^{t-1} - 1\Big)_R$$

According to the assumption, all but one element of the equation above are known or can be easily computed. Only R^{t-1}_R is unknown. And L^{t-1} can also be computed when R^{t-1}_R is available. All 2^{40} possible values can be enumerated with brute force. SHA-1 property leads to the fact that equality always holds for the correct value. And for the remaining $2^{40}-1$ it holds with probability 2^{-40}. As a consequence we expect to have at most constant amount ($\mathcal{O}(1)$) of solutions to the above equation, only one of them is the desired R^{t-1}_R.

An attack with an overhead of 2^{23}. From the general form one can derive the following equation (for details see source):

$$\mathsf{R}_R^{t-1} = \text{SHA-1}\Bigg(\Big((\mathsf{S}^t - \mathsf{R}^t - 1) \oplus \mathsf{RC}^{t-1} \oplus \mathsf{R}^{t-1}\Big) - \mathsf{R}^{t-1} - 1\Bigg)_R$$

To shorten the notation denote $Y = (\mathsf{S}^t - \mathsf{R}^t - 1) \oplus \mathsf{RC}^{t-1}$, as every element here is known or can be easily computed. Let r_i denote i-th less significant bit of R^{t-1}.

$$\begin{aligned}
&Y \oplus \mathsf{R}^{t-1} - \mathsf{R}^{t-1} - 1 = \\
&\qquad = \sum_{i:Y_i=0} 2^i r_i + \sum_{i:Y_i=1} 2^i(1-r_i) - \sum_{i=0\ldots159} 2^i r^i - 1 \\
&\qquad = \sum_{i:Y_i=0} 2^i r_i + \sum_{i:Y_i=1} 2^i - \sum_{i:Y_i=1} 2^i r_i - \sum_{i=0\ldots159} 2^i r^i - 1 \\
&\qquad = \sum_{i:Y_i=0} 2^i r_i + \sum_{i:Y_i=1} 2^i - \sum_{i:Y_i=1} 2^i r_i - (\sum_{i:Y_i=0} 2^i r_i + \sum_{i:Y_i=1} 2^i r_i) - 1 \\
&\qquad = Y - 2 \cdot \sum_{i:Y_i=1} 2^i r_i - 1 \\
&\qquad = Y - 2 \cdot (\mathsf{R}^{t-1} \bigwedge Y) - 1
\end{aligned}$$

Here, $\bigwedge$ denote bitwise "and". Overall,

$$\mathsf{R}_R^{t-1} = \text{SHA-1}(Y - 2 \cdot (\mathsf{R}^{t-1} \wedge Y) - 1)_R \qquad (5.1)$$

It is enough now to enumerate only those bits of R_R^{t-1} that effect output. In case there are l such bits (Hamming weight of Y is l), the expected overhead of the attack is $\Sigma_{l=0}^{40} 2^l \Pr[|i : Y_i = 1| = l] = \Sigma_{l=0}^{40} 2^l \binom{40}{l} 2^{-40} = 2^{-40} \Sigma_{l=0}^{40} 2^l \binom{40}{l} = (3/2)^{40} \approx 2^{23}$.

Sum $\Sigma_{l=0}^{40} 2^l \binom{40}{l}$ is computed by Binomial theorem: $\Sigma_{l=0}^{40} \binom{40}{l} 2^l 1^{40-l} = (1+2)^4 0$

Conclusions on Windows attack. The effect of an attack on a CryptGenRandom is the lack of forward and backward security: an adversary who has access to the state of the generator at time t can easily calculate previous and subsequent states and output until the internal state is updated using system entropy. Computing all states and outputs from time t to $t+k$ requires $O(k)$ work, while computing candidates for states and outputs from t to $t-k$ can be done with $O(2^{23}k^2))$ work. An attacker with access to the state at time t can use this knowledge to determine all generator states in an "attack window" that extends from the last state update before the attack to the first update after the attack.

In combination with an attack on user memory space, when an adversary can break into the address space of a specific application, for example with a buffer overflow attack, the above weaknesses become critical. That is a difficult task, but once it is completed, the adversary gains the current state of PRNG and consequently all outputs between rekey.

As mentioned above, CryptGenRandom keeps a unique instance for each process. Access to the internal state of the generator at a certain point in time allows an adversary to predict 128 KB of its output data between updates of entropy.

For processes with low consumption of randomness with can be critical. For example, according to the source, a single browser process would be able to perform 600-1200 SSL connection with this amount of entropy.

We need to note that SSL protocols were deprecated (the latest version SSL 3.0 was deprecated in June 2015) and replaced by more modern TLS. Nowadays standard is TLS 1.2 (estimated to stay secure approx. until 2026) and TLS 1.3. But primitives, on which the generator is based, also are not trusted anymore. RC4 stream cipher was deprecated by NIST in 2014 and the SHA-1 hash function was formally deprecated by NIST in 2011.

5.2 ISAAC

A cryptoanalytical attack on ISAAC was developed by M. Pudovkina in 2001 [36].

In the paper explaining the attack author firstly identifies non-trivial connections between variable states in the form of equality. It is stated that generally, similar PRNG can be generalised to any size by the change in parameters' length. Author states that using their method one can effectively propagate knowledge or guess of current state of generator to previous and future states.

Overall time complexity of an attack is approximetely a square root of brute force attack, where all possible internal states should be tested.

Note, that the study is called "a known plaintext cryptoanalytical attack on ISAAC keystream cipher" as it is expected that output of the generator is used for encryption directly:

$$\text{ciphertext} = \text{plaintext} \oplus \text{ISAAC(IV)}$$

To verify that correctness of key (or initial value) guess, one needs to compare keystreams obtained from revealed key and from real. Keystream can be obtained from sum of ciphertext and plaintext. Therefor it is a known plaintext attack on a keystream cipher, but in term of attack on PRNG, it is a known output attack.

5.3 Schneier Cryptanalytic Attacks on Pseudorandom Number Generators

5.3.1 The ANSI X.17 PRNG

A brief recollection of the design: T_i is a temporary value produced from current timestamp. K is a permanent constant DES key of the generator. V_i is an internal state.

$$\begin{array}{ll} \text{process the timestamp:} & T_i = E_K(\text{timestamp}) \\ \text{produce pseudorandom bits:} & \text{output}[i] = E_K(T_i \oplus V_i) \\ \text{update the internal state:} & V_{i+1} = E_K(T_i + \text{output}[i]) \end{array}$$

Direct Cryptanalytic Attack. To perform direct Cryptanalytic Attack requires a cryptoanalysis of DES cipher. About 20 years ago such task was considered not effective and too costly - no benefit for an attacker. Nowadays, however, DES key might be recovered within few days. Similarly, if triple-DES is used, its key would require cubical amount of time for an attack. However, even a period of a week is extremely unreliable, knowing that the key does not change until the end of the session.

An important detail is hidden behind the fact that ANSI X.17 standard suggests a way to produce DES keys. So, important detail is that not only generator is vulnerable, but also cipher for which the key and initial vector are produced. Also, outputting keys means that the PRNG outputs are almost never directly seen. That implies that intruder is not expected to analyse pseudorandom sequence and hence to distinguish it from truly random.

Input-Based Attacks. Assuming a 64-bit block size, PRNG has a weakness with respect to replayed-input attacks. If attackers can manipulate the system time, they will be able to distinguish pseudo-random output from truly random numbers after generating approximately 2^{32} 64-bit output. That is because in a truly random sequence, a collision is expected to happen after about 2^{32} outputs. However, with a constant timestamp, a collision in X9.17 is expected to require approximately 2^{63} outputs.

State Compromise Extension Attacks. According to the source the X.17 PRNG does not properly recover form state compromise. [21]. That is knowing internal state and the generator's key K, an attacker who knows basic properties of timestamps used can reproduce previous outputs and predict future with appropriate effort.

The Permanent Compromise Attack: Deriving the Internal State from Two Outputs. Assume an attacker who knows the generator's key K and gained access to two consecutive outputs $output[i, i+1]$. For example, an intruder is communicating with our vulnerable system pretending to be the third party willing to communicate through the secure channel and during the key exchange receives generated sequence. An intruder's goal is to reveal the internal state $V[i+1]$. The brute force attack would require a 64-bit search through all possible $V[i+1]$ values. However, it is possible to break through more efficiently.

As stated in the source, it is a reasonable to assume, that an attacker might gather the timestamp value to the high precision. Knowing time when generator used a timestamp to about the nearest second leaves just about about 10 bits of uncertainty - milliseconds. All assumptions above allow to mount a meet-in-the-middle attack that requires about 2^{11} trial encryption runs. From the PRNG design we can derive 2 equations to describe an internal state at time $i+1$:

$$V_{i+1} = D_K(\text{output}[i+1]) \oplus T_{i+1}$$
$$V_{i+1} = E_K(\text{output}[i] \oplus T_i)$$

Trying all possible T_i and all possible T_{i+1} 2 lists are created: a list of possible V_{i+1} value depending on the value of T_i and a list of possible V_{i+1} value depending on the value of T_{i+1}. Value that is available in both is the wanted internal state.

The Iterative Guessing Attack. Under the common assumption for all State Compromise Extension Attacks of revealed generator's key K assume an impostor knows previous internal state V_i and output$[i+1]$ or at least a function of an output (e.g. a plain text and it's corresponding encrypted with key output$[i+1]$ cipher text).

The aim is to reveal internal state V_{i+1}. Should an impostor succeed, then it will be possible to repeat an attack once function of output$[i+2]$ is available. Basically, it allows to recreate future outputs as long as a little additional information is available for every of them.

The attack is designed in the following way:

- Using known key K try to guess T_i (an entropy of the timestamp can be estimated to be low similar to previous example). This leads to about 2^{10} guessed candidates for the internal state V_{i+1}

$$V'_{i+1} = E_K(E_K(T_i \oplus V_i) \oplus T_i)$$

- For every candidate V'_{i+1} for all appropriate T_{i+1} produce an output and a function of it.
- The combination of V'_{i+1} and T_{i+1} that leads to the eavesdropped function value is the value that an intruder is looking for.

While revealing of two consecutive outputs is a strong assumption (given that outputs are encryption keys), a function of output might be much easier to access for a potential attacker.

Backtracking. To reveal earlier states and outputs attacker does not need much of an effort:

Assuming that V_i is known and that T_{i-1} can be guessed from about 2^{10} options, there are 2^{10} candidates for a previous output:

$$\text{output}[i-1] = D_K(V_i) \oplus T_{i-1}$$

and the same amount for previous internal state:

$$\begin{aligned} V_{i-1} &= D_K(\text{output}[i-1]) \oplus T_{i-1} \\ &= D_K(D_K(V_i) \oplus T_{i-1}) \oplus T_{i-1} \end{aligned}$$

The correct result among this set can only be recognized by comparing with eavesdropped function of output$[i-1]$.

Meet-in-the-middle. The backtracking and iterative guessing attacks work in a similar way: the entropy introduced in every cycle of generation is brute forced and the correct parameters are then identified by comparing possibles results with the only correct one. The meet-in-the-middle uses similar approach. Assume, nonconsecutive internal states are known: V_{i-c} and V_{i+c}. Every round of generating pseudorandom sequence introduces a little bit of entropy by using a timestamp, let us assume the same 2^{10} options for each.

Then, starting at V_{i-c}, we can get $2^{10\cdot c}$ candidates for V_i. And, starting at V_{i+c}, we can backtrack to $2^{10\cdot c}$ candidates for V_i. The correct value of V_i must be in the interception these 2 lists, however there might be more than 1 unique combination of $[T_{i-c}, \ldots, T_{i+c}]$ timestamps that lead from V_{i-c} to V_{i+c}. According to Schneier [21] this attack for known states V_i and V_{i+8} might return 2^{16} such combinations.

Summary. In general terms, ANSI X9.17 generator was safe, and even complete exposure of the key (it would seem that this is already the end) leaves some difficulties, without overcoming which it is impossible to achieve a completely dominant position. The generator could still be used nowadays after replacing the primitive function (for example with the supposedly more reliable AES). However, if the key and/or internal state is exposed, the generator may never recover without a full reboot.

5.3.2 The DSA PRNG

A brief recollection of the design: x_i is an internal state of the generator, W_i is an optional input.

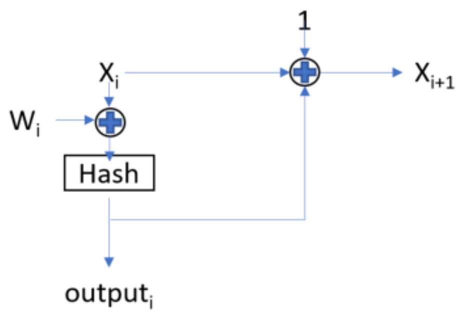


Figure 5.1: DSA scheme (again)

Direct Cryptanalytic Attack. According to the standard, the primitive used is a 'one-way function'. Suggested choice are based on either a hash-function SHA-1 or on DES primitive. However, one can employ more modern primitives to ensure resistance against cryptanalytical attacks. SHA-1, similar to DES, is not concidered secure since at least 2005 when first collision attacks were announced publically. [26]

Input-Based Attacks. Should attackers have the ability to control inputs, they are able to force PRNG repeat the same output unlimited time. It can be reached by enforcing W_i optional values:

$$\begin{aligned} W_i &= W_{i-1} \oplus \text{output}[i-1] \oplus 1 (\text{mod } 2^{160}) \\ (&= W_{i-1} \oplus X_{i-1}) \end{aligned}$$

State Compromise Extension Attacks. Generally, state compromise is not the termination of security for the DSA PRNG. Due to the ability to introduce an unlimited amount of entropy into the system, it can recover over time. However, with deliberate, careful steps, an attacker can maintain control for a very long time.

Leaking input effect. This is not an attack, but a disadvantage of an algorithm. Each new input brings some external influence on the state of the generator. The disadvantage is that this influence is not multiplied, but is only replaced by the next one, since the attacker does not need to hold and process a lot of previous external data - it is enough for him to remember the previous output of the generator, which contains the resulting force of all previous inputs.

The authors suggest, that this problem can be solved with a simple change: output shall not give a feedback into the next internal state, that is $X_{i+1} = 1 \oplus X + i \oplus W_i$. Authors state that in this case leaking input effect would not appear. Nevertheless, W_i is an optional input and it might not be given every round. Then such a change would be more of a threat than an improvement. Overall, the solution offered in the source [21] seem to work the best. The change suggested is define the next internal state in the following manner:

$$X_{i+1} = X_i + \text{hash}(\text{output}[i] + W_i) \bmod 2^{16}$$

The Iterative Guessing Attack. An attack relies on the vulnerability introduced with a poor choice of W_i values. Should attackers gain a knowledge of the internal state and knowledge of W_i pattern they can use both in an iterative guessing to reproduce the next internal state. For example, when this optional input has only limited amount of entropy bits, an attacker might restrict a set of possible values and instead of brute force walk through a much smaller set of inputs.

The attack itself expects that for every possible W_i using known X_i the output is produced and used to be compared with an eavesdropped value of real function of the output such as a DSA signature made with output$[i]$ as its secret parameter value.

Backtracking. Backtracking is only possible when the previous output is known along with the current internal state Overall revealing the previous state does not seem to bring some added value for an attacker who holds previous output. But it still might come in handy under certain circumstances.

Summary. While the DSA standard's PRNG proves robust for its intended purpose — DSA signature parameter generation — it falls short as a multipurpose cryptographic PRNG. Its suboptimal input handling and slower recovery from state compromise limits its effectiveness in general cryptographic applications.

5.3.3 The RSAREF PRNG

A brief recollection of the design: C_i - the internal state

- the state can be updated with an optional input X:

$$C_{i+1} = C_i + \text{MD5}(X) \bmod 2^{128}$$

- Output is produced as a hash of current state:

$$\text{output}_i = \text{MD5}(C_i) \bmod 2^{128}$$
$$C_{i+1} = C_i + 1 \bmod 2^{128}$$

Direct Cryptanalytic Attack. Similar to earlier mentioned SHA-1 and DES, MD5 is not considered secure nowadays. There are publically available libraries of precomputed hashes and several attacks are designed and published. Assuming a modern secure hash function is empoyed instead of MD5 there are still some options:

Partial Precomputation. To simply identify internal state of the generator seeing its output one may precompute output of the generator for every t-th value of the counter. Once such value is met in the generated sequence, internal state can be considered compromised. This type of attack is highly impractical. Attacker, who expects to eavesdrop 2^{32} outputs must precompute 2^{96} hash values and search through all hashed for every output.

Timing attack. Increment of the internal state is done in a way that allows optimise runtime of the algorithm:

```
  /* increment state */
  for (i = 0; i < 16; i++)
    if (randomStruct->state[15-i]++)
      break;
```

The code leaks some information about the resulting 128-bit counter by how many 8-bit add operations the computer must execute.

Input-Based Attacks. When a control over inputs is available, it is possible, to pass the same X prior to every iteration of generation such that MD5(X) has it's least significant bits set to all 1. The idea is that after output counter will be always increment by 1 and another constant number with 1 in all least significant bits - when combined, result of 2 increments is an addition of number that ends with all 0. Described strategy forces the PRNG to cycle much faster, because the low-order n bytes of the counter are fixed, however still hidden from an attacker.

Another way to affect the internal state by inputs is to feed 2 precomputed chosen inputs. In no external (not controlled by an attacker) input was provided during j sequential outputs, attacker can reset internal state to the same as j outputs ago. It requires availability of 2 values X_1 and X_2 such that MD5(X_1)+ MD5(X_2) $= -j$. Then by passing two chosen inputs after j outputs an attackers resets the state to

$$C_i + j + \text{ MD5}(X_1) + \text{ MD5}(X_2) = C_i + j - j$$

State Compromise Extension Attacks. *Backtracking.* Same as forward simulation, backtracking does not appear difficult for an intruder. Knowing internal state (the value of counter), it is enough to increment it by 1 to simulate future outputs, and decrement by 1 to reproduce previous outputs.

However backtracking and forward simulation will both work only on the interval, between two unknown inputs were passed to update the state.

Iterative guessing attack. Unless an unpredictable input was passed to update the state of PRNG, an attacker can compute future states of PRNG and predict it's next outputs. Otherwise, knowledge of the state will be lost.

Summary. Overall the PRNG was safe to use until it's primitive - MD5 - was strong. The RSAREF 2.0 PRNG is susceptible to chosen-input attacks, potentially leading to short cycles. It is also vulnerable to chosen-input timing attacks that can expose its secret state. Additionally, iterative guessing and backtracking attacks can enable an attacker to extend their knowledge of the secret state both backward and forward in time.

With a little improvements, PRNG of such design can remain relatively safe even nowadays.

- Firstly, more modern one-way function shall be used instead of deprecated MD5.
- To limit abilities of the potential intruder inputs used for an update of counter shall be combined with some source of entropy that can not be affected from outside. Authors suggest appending current timestamp and/or a counter.
- designing the system, technical implementation should pay attention to prevention of chosen-input attacks and timing attacks.

6. Congruential generators

After cryptanalytic attacks on various generators, we now consider congruential generators. These generators are a fundamental class of pseudorandom number generators that are widely studied and used due to their mathematical simplicity and efficiency. However, analyzing and attacking congruential generators poses interesting and challenging problems.

Congruential generators, including linear congruential generators (LCGs) and multiple recursive generators (MRGs), produce sequences of numbers based on linear congruences. Attacks on congruential generators often rely on advanced mathematical techniques, especially lattice pruning algorithms. Lattice pruning algorithms such as LLL play a significant role in these attacks, making the analysis of congruential generators a fascinating and challenging area of research.

A linear congruential generator (LCG) is an algorithm proposed by Derrick H. Lehmer [3]. The algorithm produces a sequence of pseudo-random numbers calculated using a linear equation. Generator has a state a_n and a linear recurrent equation that defines a transition to the next state:

$$a_{n+1} = (c \cdot a_n + b) \pmod{m}, \tag{6.1}$$

where b and m are natural numbers ($0 < b < m$), $c < m$ is an integer. The state itself is an output in each round of generation. As such a generator has vulnerabilities and cannot be considered secure. However, the generator can be improved to amend it, and this can be done in two ways. One is to hide a significant fraction of the internal state, another is to apply a recurrence of higher rank.

A truncated linear congruential generator (TLCG) is an extension of LCG. It functions in exactly the same way but truncates internal state to only few bits prior to output.

As stated in the source [4] LCG and TLCG are the simplest instances of a general congruential generator that is defined as follows

$$a_{n+1} = \sum_{j=1}^{k} c_j \cdot \varphi_j(a_n, \ldots, a_0) \pmod{m}, \tag{6.2}$$

where modulus m, index k, and coefficients $c_1, \ldots, c_k$ as well as states a_n, $n > 0$, are integers.

6.1 Attacking Congruential Generators

This section is based on the study of Boyar [42] and Krawczyk [41], both summarised in research of Odlyzko[4].

Assumption of the attack is that the analyst or attacker knows structure of PRNG (that it is a congruential generator as in (6.2)) and knows the functions φ_j. Coefficients c_j and/or modulus m are hidden.

The task is as follows: given the outputs $a_1, a_2, \ldots, a_n$, find the output a_{n+1}. Note, that from definition of generator to run a cycle of generator and produce first output, some initial values (IV) are required. This IV might be strategically hidden by the design of an attacked system. If that is the case, an attacker would require to assume another feature of the system: that functions φ_j do not operate on all previous states of generator including IV, but only on a limited subset of last few outputs. Then to perform an attack one do have to firstly eavesdrop first n outputs to have arguments for the functions φ_j.

In the following text, without loss of generality assume that IV is known to an attacker. Odlyzko defines that an attack is successful if it's runtime is polynomial in $\log_2 m$ and k (number of functions φ_j).[4, 41]

To describe an attack, first introduce the notation. In the forthcoming discussion to make reading easier, we will use an index j to denote the iteration of functions φ_j (we still assume that there are k of those), while an index i will denote the iterative outputs of the generator a_n. A total of n outputs (if IV are available, they are uncounted in this number) are known and the state a_{n+1} is to be guessed by an attacker.

Let

$$B_{n+1} = \begin{pmatrix} \varphi_1(a_n, \dots, a_1) \\ \varphi_2(a_n, \dots, a_1) \\ \vdots \\ \varphi_k(a_n, \dots, a_1) \end{pmatrix}. \tag{6.3}$$

Note, that indexes are shifted by one on different sides of equation. It comes from the fact, that B_{n+1} defines generator state a_{n+1} using a matrix multiplication as follows:

$$c \cdot B_{n+1} = \begin{pmatrix} c_1 | c_2 | \cdots | c_k \end{pmatrix} \cdot \begin{pmatrix} \varphi_1(a_n, \dots, a_1) \\ \varphi_2(a_n, \dots, a_1) \\ \vdots \\ \varphi_k(a_n, \dots, a_1) \end{pmatrix} = a_{n+1} \tag{6.4}$$

Predicting coefficients c_j would be a first idea for any deterministic attack. However, Krawczyk[41] and Boyar [42] offer an alternative solution. Instead of guessing coefficients, main idea behind the the Krawczyk's and Boyar's attacks is that for all but possibly k values of n there exist integers y_i, such that $y_{n+1} \neq 0$ and

$$y_{n+1} B_{n+1} = \sum_{i=0}^{n} y_i B_i. \tag{6.5}$$

According to assumption of an attack, all previous outputs are known as well as the functions φ. Hence, an attacker can compute the vectors B_i. To make a prediction of the future state, an attacker tries to find a solution to

$$B_{n+1} = \sum_{i \in \mathrm{Ind}_n} y_i B_i, \tag{6.6}$$

where Ind_n is a set of indexes $i \leq n$ such that $[B_i]_{i \in \mathrm{Ind}_n}$ is the largest linearly independent set of vectors.

There are 2 possible outcomes:

- Solution of (6.6) does not exist. It means that B_{n+1} is independent of previous vectors and attacker adds it to the set of independent vectors.

- There is a solution $\hat{y}$. In that case, a prediction $\hat{a}_{n+1}$ is computed as:

$$\hat{a}_{n+1} = \sum_{i \in \mathrm{Ind}_n} \hat{y}_i a_i \tag{6.7}$$

The above step is the same when m is not still guessed (then computations are done over the field $\mathbb{Q}$) and when some guess $\hat{m}$ of the modulus m has been predicted already (then computations are done over the ring $\mathbb{Z}_{\hat{m}}$).

Once prediction $\hat{a}_{n+1}$ is made, true value a_{n+1} is recovered, it can be eavesdropped. The prediction and true value might not equal. Then the first prediction $\hat{m}$ of the modulus m is made as follows

$$\hat{m} = |d\hat{a}_{n+1} - da_{n+1}|, \tag{6.8}$$

where d is the LCM of denominators of values in $\hat{y}$ (the first prediction is made over $\mathbb{Q}$).

The update of prediction $\hat{m}$ is defined as

$$\hat{m} = gcd(\hat{m}, \hat{a}_{n+1} - a_{n+1}). \tag{6.9}$$

We summarise the attack in Algorithm 8 below.

While computations in the Algorithm 8 are simple and straight-forward, their result is not clear. Possible questions are hidden behind the need to prove following facts:

1. $|d\hat{a}_{n+1} - da_{n+1}|$ is a multiple of modulus m.
2. $gcd(\hat{m}, \hat{a}_{n+1} - a_{n+1})$ is a non-trivial multiple of m.
3. the run-time of an attack is polynomial in $\log m$ and k.

First two ensure that algorithm returns correct values. And the third statement implies that algorithm can finish and is efficient.

Proof. (1)

By definition of the Congruential Generator it holds.

$$a_{n+1} = c \cdot B_{n+1} \pmod{m}. \tag{6.12}$$

Then, by definition of $\hat{a}_{n+1}$:

$$\begin{align}
\hat{a}_{n+1} &= \sum_{i \in \mathrm{Ind}_n} \hat{y}_i a_i \tag{6.13}\\
&\equiv \sum_{i \in \mathrm{Ind}_n} \hat{y}_i \cdot c \cdot B_i \pmod{m} \qquad \text{by (6.12)} \tag{6.14}\\
&= c \cdot B_{n+1} \qquad \text{as } \hat{y} \text{ solves (6.6)} \tag{6.15}\\
&\equiv a_{n+1} \pmod{m} \tag{6.16}
\end{align}$$

So, $|\hat{a}_{n+1} - a_{n+1}|$ is a multiple of m and a multiplication by d is required to ensure that computations are done over $\mathbb{Z}$ rather than $\mathbb{Q}$ (because $\hat{y}_i$ might be a fraction).

□

Proofs of 2 and 3 rely on many interim claims from sources, therefor will not be stated here.

Algorithm 8 Attack on Congruential Generator

Initial Inputs: $a_1, \dots, a_n$ (first outputs), $\varphi_1, \dots, \varphi_k$ (definitions of functions)
Dynamic Inputs: a_i - values produced by the generator after each attacker guess $\hat{a}_i$
Outputs: modulus m, predictions of future generator outputs $\hat{a}_i$

```
 1: for  $i \in (1, \dots, n)$ do:
 2:
        $$B_i = \begin{pmatrix} \varphi_1(a_i, \dots, a_1) \\ \varphi_2(a_i, \dots, a_1) \\ \vdots \\ \varphi_k(a_i, \dots, a_1) \end{pmatrix} \quad (6.10)$$
 3: end for
 4: $\mathrm{Ind}_n \leftarrow \{i\}$, $i \leq n$ such that $[B_i]_{i \in \mathrm{Ind}_n}$ is the largest linearly independent set
    of vectors,
 5: $\hat{m} \leftarrow inf$,
 6: $i \leftarrow n + 1$                              ▷ current sequential number of output

 7: while $\hat{m} = inf$ do          ▷ Phase 1 (before first guess of modulus is made)
 8:     solve for $y_i \in Q$:
        $$B_{n+1} = \sum_{i \in \mathrm{Ind}_n} y_i B_i \quad (6.11)$$
 9:     if solution does not exist then
10:         $\mathrm{Ind}_n \leftarrow \mathrm{Ind}_n + \{i\}$
11:     else
12:         $\hat{a}_{i+1} \leftarrow \sum_{i \in \mathrm{Ind}_n} \hat{y}_i a_i$
13:         $d \leftarrow \mathrm{lcm}(\text{denominators of } y_i)$
14:         $\hat{m} \leftarrow |d\hat{a}_{n+1} - da_{n+1}|$
15:     end if
16: end while

17: while $\hat{m} \neq m$ do                                        ▷ Phase 2
18:     Solve for $y_i \in Z_{\hat{m}}$: $B_{i+1} = \sum_{i \in \mathrm{Ind}_n} y_i B_i \pmod{\hat{m}}$
19:     if solution does not exist then
20:         $\mathrm{Ind}_n \leftarrow \mathrm{Ind}_n + \{i\}$
21:     else
22:         $\hat{a}_{i+1} \leftarrow \sum_{i \in \mathrm{Ind}_n} \hat{y}_i a_i$
23:         if $\hat{a}_{i+1} \neq a_{i+1}$ then
24:             $\hat{m} \leftarrow gcd(\hat{m}, \hat{a}_{i+1} - a_{i+1})$
25:         end if
26:     end if
27: end while
```

6.2 Multiple Recursive Generator

A multiple recursive generator (MRG) are an instance of Congruential Generator, and it generalizes the Linear Congruential Generator. The linear recurrence used to define MRG has a higher order, the next state a_{i+n} depends on multiple previous states. The following definition is a necessary prerequisite to define a Multiple Recursive Generator.

Definition 30 (linear recurrent sequence)**.** [6, def 1] The sequence $\underline{a} = (a_i)_{i \geq 0}$ over $\mathbb{Z}_m$ that satisfies a recurrent relation

$$a_{i+n} = (c_{n-1} \cdot a_{i+n-1} + c_{n-2} \cdot a_{i+n-2} + \cdots + c_0 \cdot a_i) \bmod m \tag{6.17}$$

is called *a linear recurrent sequence* of order n over $\mathbb{Z}_m$.

Polynomial $f(x) = x^n - c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \cdots + c_0$ is called a characteristic polynomial of sequence $\underline{a}$. A set of all sequences generated by polynomial $f(x)$ over $\mathbb{Z}_m$ is denoted $G(f(x), m)$.

MRG is a generator that produces linear recurrent sequence. Note, that LCG defined by (6.1) is also an instance of MRG. In terms of the above definition it generates a linear recurrent sequence of order 1.

Similarly to linear congruential generators, internal state of MRG can be additionally truncated prior to outputting it. Assuming, $\underline{a} = (a_i)_{i \geq 0}$ is a linear recurrent sequence, let $k = \lceil \log(m) \rceil$ be a length of $(m-1)$, the greatest number over $\mathbb{Z}_m$. Truncated output y_i consists of proportion α of high-order bits of the corresponding a_i. Hence, a_i can be divided into secret part z_i and output y_i:

$$a_i = 2^{(1-\alpha)k} y_i + z_i \tag{6.18}$$

MRGs are widely used in cryptography. For example, a stream cipher ZUC is an instance of it. It serves as a cryptographic primitive for the 3GPP mobile standards 128-EEA3 and 128-EIA3, included in the Long Term Evolution (4G LTE) standards starting in Release 11.

In its public specification, ZUC is defined as shift register with similar principle as LFSR operating on 32-bit long words. The recurring equation for ZUC is:

$$\begin{aligned} a_{i+16} = 2^{15} \cdot a_{i+15} + 2^{17} \cdot a_{i+13} + 2^{21} \cdot a_{i+10} \\ +2^{20} \cdot a_{i+4} + (1 + 2^8) \cdot a_i \pmod{2^{31} - 1}. \end{aligned} \tag{6.19}$$

Note, that $2^{31}-1$ is not only a prime, but also the greatest value a signed word can represent in a computer memory. Ensuring, that operation modulo $2^{31}-1$ are computationally efficient and a maximum period of MRG is the highest possible (for prime modulus m, the period is $m^n - 1$). When used in practice, internal state is further transformed in so-called bit-reorganization layer and a non-linear function. However, for the following cryptoanalytical attacks, these were ignored, authors only used ZUC's MRG to test performance of their theoretical developments.

As MRGs are instances of general Congruential Generators and they generalize LCGs, some of properties are shared. And techniques of cryptoanalysis of different congruential generators not only share similar patterns but are directly used to derive those more general. For example, authors of study about MRGs [6] refer to Boyar's and Stern's studies of LCGs and truncated LCGs [9, 10].

6.3 Attacking truncated MRG

The attack is described as the following mathematical problem. Let $\underline{a} = (a_i)_{i\geq 1}$ be a linear recurrent sequence, let y_i denote a part of high-order bits of a_i. Given first N truncated elements of sequence $\underline{a}$ recover modulus m, coefficients $c_0, \dots, c_{n-1}$ and initial state $a_0, \dots, a_{n-1}$.

Two articles study the topic. Firstly, Sun et al. offered an attack on MRG in [6]. Later, Yu et al. offered a significant performance improvement to it in [5]. Both studies primarily concentrate on exploring methods for recovering coefficients $c_0, c_1, \dots, c_{n-1}$ and modulus m. The authors propose using results of other studies to restore initial values. We will return to it in the end of the attack description.

Here's a step-by-step breakdown of the original method, also referred to as "Sun-Zhu-Zheng's" by the names of authors. It consists of 5 stages: searching linear relations, constructing congruence equations and firstly recovering modulus, than recovering coefficients $c_0, \dots, c_{n-1}$ and finally recovering the initial state of generator.

Sun-Zhu-Zheng's Method

1. Searching linear relations: The goal is to find a set of linear relations for a segment of the sequence $\underline{a}$. This is achieved by using a lattice reduction algorithm to find linear relations with small coefficients.

Firstly, a lattice reduction algorithm is executed for r segments of truncated to length t elements of sequence $\underline{a}$ ($r > t > n$), denoted as $Y_i = (y_i, y_{i+1}, \dots, y_{i+t-1})$. The parameters r and t in practice are selected experimentally.

Clearly, there exist integer coefficients $\zeta_0, \dots \zeta_{r-1}$ such that $\sum_{i=0}^{r-1} \zeta_i Y_i = 0$. The statement holds true because the number of vectors exceeds their dimension; thus, they cannot be linearly independent. Additionally, these coefficients are bounded $|\zeta_i| \leq B$ [6, lemma 6], [7], where B depends on parameters r, t, and proportion of high-order bit α.

Then, the norm of vector $\zeta = (\zeta_0, \dots \zeta_{r-1})$ is bounded above by $\sqrt{r}B$, as each its element $|\zeta_i| \leq B$.

For a parameter $K = \sqrt{r}2^{(r-1)/2}B$ define a lattice

$$L = \begin{pmatrix} KY_0 & KY_1 & \cdots & KY_{r-1} \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}, Y_i \text{ are treated as columns here.} \tag{6.20}$$

It holds that $W = (\underbrace{0, \dots, 0}_{t\text{-times}}, \zeta_0, \dots \zeta_{r-1})^T$ is in lattice L, at the same time the shortest vector in L has norm $\lambda_1(L) \leq ||W|| \leq \sqrt{r}B$. Note, that lattice L could be defined without multiplication of Y_i by K and it still would include W. Also, a motivation of these practise is not explained in the source, we can assume that it is a practical trick to guide the LLL algorithm towards finding a short vector of a desired shape. Artificial increase of leading parts of base vectors can be used to control the reduction focus and, hence, ensure that reduced base includes vectors that start with leading zeros.

Let $W_0, \dots, W_{r-1}$ be the LLL reduced basis of L. For every vector of basis $W \in W_0, \dots, W_{r-1}$ holds

$$||W|| \leq 2^{(r-1)/2}\lambda_1(L) \leq 2^{(r-1)/2}\sqrt{r}B \leq K. \tag{6.21}$$

Let W_0 has the form $(0, 0, \dots, 0, \eta_0, \dots, \eta_{r-1})$ such that

$$\sum_{i=0}^{r-1} \eta_i Y_i = 0. \tag{6.22}$$

Authors claim that there are usually more than one vectors W_i of form $(0, 0, \dots, 0, \eta_0, \dots, \eta_{r-1})$ [6, Remark 2]. Therefor a single call to lattice reduction algorithm can produce multiple solutions.

However, the purpose of attack is to recover states a_i. While the above linear relations hold only for truncated outputs y_i. Indeed, the desired relation is $U = \sum_{i=0}^{r-1} \eta_i A_i = 0$, where $A_i = (a_i, a_{i+1}, \dots, a_{i+t-1})$. Note, that relation $a_i = 2^{(1-\alpha)k} y_i + z_i$ (6.18) and definition of coefficients $\eta_0, \dots \eta_{r-1}$ together imply

$$U = \sum_{i=0}^{r-1} \eta_i A_i - \underbrace{\sum_{i=0}^{r-1} \eta_i Y_i}_{\text{equals } 0} = \sum_{i=0}^{r-1} \eta_i Z_i. \tag{6.23}$$

Sun [6] show that for an appropriate choice of parameters r and t, relation $U = 0$ will be valid. The core idea is first to construct a lattice such that U belongs to it, and then the previous equation holds if U is shorter than the shortest non-zero vector of this lattice.

2. Constructing congruence equations: Using coefficients $c_0 \dots c_{n-1}$ that define linear recurrence, let

$$Q = \begin{pmatrix} 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-2} \\ 0 & 0 & \cdots & 1 & c_{n-1} \end{pmatrix}. \tag{6.24}$$

Then, recurrence relation can be expressed as follows:

$$(a_{i+j}, a_{i+j+1}, \dots, a_{i+j+n-1}) = (a_i, a_{i+1}, \dots, a_{i+n-1})Q^j \pmod{m}. \tag{6.25}$$

Let $\begin{pmatrix} q_{j,0} \\ \vdots \\ q_{j,n-1} \end{pmatrix}$ be the first column vector of Q^j. Then we can express element a_{i+j} as follows:

$$a_{i+j} = \sum_{l=0}^{n-1} q_{j,l} a_{i+l} \pmod{m}. \tag{6.26}$$

Linear relations found in the first step combined with the above expression implies that the following congruence could hold.

$$\begin{pmatrix} a_0 & a_1 & \cdots & a_{n-1} \\ a_1 & a_2 & \cdots & a_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{t-1} & a_t & \cdots & a_{t+n-2} \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{n-1} \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \pmod{m}, \tag{6.27}$$

where

$$g_i = \eta_i + \sum_{j=n}^{r-1} \eta_j q_{j,i} \tag{6.28}$$

Based on source [8] the following theorem is proven [6]: Let $f(x)$ be a primitive polynomial over $\mathbb{Z}/(m)$ and let $\underline{a} \in G(f(x), m)$, then the congruence stated above has only the trivial solution ($g_i \equiv 0 \pmod{m}$).

An attack offered by Sun [6] is primarily designed to address ZUC MRG used in LTE standard. This and the most of others MRG do use primitive polynomials. However limiting an attack to standards of ZUC would mean that next two steps are not required: modulus and coefficients are an integral part of the cipher.

3. Recovering the modulus m: Core idea behind recovering the modulus is to form a lattice that has a determinant m from construction (because it exploits the m). Then, we want to investigate a sublattice with known vectors. Determinant of such sub lattice shall be a multiple of m.

By definition $g_i = \eta_i + \sum_{j=n}^{r-1} \eta_j q_{j,i}$. If it holds that $g_i \equiv 0 \pmod{m}$, then there exists $u_i \in \mathbb{Z}$ such that

$$\eta_i = u_i m - \sum_{j=n}^{r-1} \eta_j q_{j,i} \tag{6.29}$$

Construct the following lattice $L(g_i)$:

$$L(g_i) = \begin{pmatrix} m & -q_{n,i} & -q_{n+1,i} & \cdots & -q_{r-1,i} \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{6.30}$$

Let $\eta(i) = (\eta_i, \eta_n, \eta_{n+1}, \dots, \eta_{r-1})^T$. Then, denoting columns of $L(g_i)$ with v, $\eta(i)$ is clearly a linear combination of column vectors of $L(g_i)$:

$$\eta(i) = u\mathrm{v}_0 + \sum_{j=n}^{r-1} \eta_j \mathrm{v}_{j-n+1} \in L(g_i). \tag{6.31}$$

Suppose, an attacker knows d distinct truncated sequences of MRG outputs. Then, following steps 1 and 2 it is possible to find at least d vectors $\eta(i)$. Denote them with j: $\eta(i)^{(j)} = (\eta_i^{(j)}, \eta_n^{(j)}, \eta_{n+1}^{(j)}, \dots, \eta_{r-1}^{(j)})^T$, for $j = 0 \dots d-1$.

The method uses the determinant of a lattice

$$M(g_i) = \begin{pmatrix} \eta_i^{(0)} & \cdots & \eta_i^{(d-1)} \\ \eta_n^{(0)} & \cdots & \eta_n^{(d-1)} \\ \eta_{n+1}^{(0)} & \cdots & \eta_{n+1}^{(d-1)} \\ \vdots & \ddots & \vdots \\ \eta_{r-1}^{(0)} & \cdots & \eta_{r-1}^{(d-1)} \end{pmatrix} \tag{6.32}$$

to recover the unknown modulus m. Lattice is firstly reduced to remove possibly not independent vectors. The resulting reduced lattice is shown to have a determinant equal to a multiple of m. Therefore by successively collecting rows of such matrix, the determinant will tend to equal m.

Authors only provide a heuristic on convergence of the method. They have shown experimentally that it works, but there is not any known mathematical proof of such convergence [6].

4. Recovering the coefficients $c_0, c_1, \ldots, c_{n-1}$**:** Once the modulus m is known, the coefficients can be recovered using the extended Euclidean algorithm.

Let $L(g_i)^*$ be the LLL-reduced lattice basis of $M(g_i)$ [6.32]. Then, one can use such a lattice to define a slightly different lattice H to find c_i. As proved in [6, Theorem 3] when modulus m is recovered correctly, lattice generated by two smallest vectors of reduced basis H is the same as the lattice generated by $(m, 0, 0, \ldots, 0)$ and $(-c_i, 1, 0, \ldots, 0)$. That means that the 2×2 submatrix in the upper left corner of reduced lattice $\begin{pmatrix} h_{0,0} & h_{0,1} \\ h_{1,0} & h_{1,1} \end{pmatrix}$ can be transformed into $\begin{pmatrix} m & 0 \\ -c_i & 1 \end{pmatrix}$.

Hence, there must exist integers u and v such that $uh_{0,1} + vh_{1,1} = 1$ and at the same time

$$c_i = uh_{0,0} + vh_{1,0} \pmod{m}. \tag{6.33}$$

As the values of u and v can be determined by the extended Euclidean algorithm, the coefficient c_i will be recovered.

5. Recovering the initial state $a_0, a_1, \ldots, a_{n-1}$**:** Since truncated outputs of MRG $y_0, y_1, \ldots y_{n-1}$ are known, recovering the initial state is equivalent to recovering $z_0, z_1, \ldots z_{n-1}$, because a_i is a concatenation of y_i and z_i. After transforming the problem into a problem of finding small integer solutions to systems of linear congruences [6] uses method described in [11].

Assume a total of d truncated outputs were eavesdropped, that is $y_0, \ldots, y_{d-1}$ are known. Recovering hidden parts $z_0, \ldots, z_{n-1}$ is sufficient to recover the initial state.

By substituting the output and hidden bits relation (6.18) into the matrix form of the recurrence equation defining the sequence $\underline{a}$ (6.26) the following congruence is obtained:

$$\begin{aligned}
a_{i+j} &= \sum_{l=0}^{n-1} q_{j,l} a_{i+l} \pmod{m},\\
2^{(1-\alpha)k} y_{i+j} + z_{i+j} &= \sum_{l=0}^{n-1} q_{j,l}(2^{(1-\alpha)k} y_{i+l} + z_{i+l}) \pmod{m},\\
\sum_{l=0}^{n-1} q_{j,l} z_{i+l} - z_{i+j} &= 2^{(1-\alpha)k} y_{i+j} - \sum_{l=0}^{n-1} 2^{(1-\alpha)k} q_{j,l} y_{i+l} \pmod{m}.
\end{aligned}$$

As we consider entire eavesdropped sequence at once, indexes are $i = 0$ and $n \leq j, \leq d-1$. Thus, there are $n-d$ congruence equations of d unknown. Denote vector of all unknown variables $z = (z_0, \dots z_{d-1})$. The congruence equations to be solved are

$$\sum_{l=0}^{n-1} q_{j,l} z_{i+l} - z_{i+j} = 2^{(1-\alpha)k}(y_{i+j} - \sum_{l=0}^{n-1} q_{j,l} y_{i+l}) \pmod{m}, \tag{6.34}$$

where $n \leq j, \leq d-1$.

To solve it, firstly introduce the following lattice formed with coefficients of the unknown variables on the left side of the above congruence equations:

$$L(m,d) = \begin{pmatrix}
m & 0 & \cdots & 0 & q_{n,0} & q_{n+1,1} & \cdots & q_{d-1,n-1}\\
0 & m & \cdots & 0 & q_{n,0} & q_{n+1,1} & \cdots & q_{d-1,n-1}\\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots\\
0 & 0 & \cdots & m & q_{n,0} & q_{n+1,1} & \cdots & q_{d-1,n-1}\\
0 & 0 & \cdots & 0 & -1 & 0 & \cdots & 0\\
0 & 0 & \cdots & 0 & 0 & -1 & \cdots & 0\\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & -1
\end{pmatrix}. \tag{6.35}$$

Denote by λ_d the d-th minimum of the lattice $L(m,d)$. The solution of (6.34) has been solved in [11, Theorem 2.1]. Where it is proved, that there exists at most one solution z such that:

$$||z|| \leq m\lambda_d^{-1} 2^{\frac{-(d-1)}{2}-1}.$$

By the nature of hidden digits, every hidden part of out put z_i is at most $2^{(1-\alpha)k}$. Hence, $||z|| \leq 2^{(1-\alpha)k}\sqrt{d}$. And when the following condition holds, hidden parts of output generated by MRG can be recovered with method in [11]

$$2^{(1-\alpha)k}\sqrt{d} \leq m\lambda_d^{-1} 2^{\frac{-(d-1)}{2}-1}.$$

Conclusion

In conclusion, this thesis has delved into the theoretical basics of Cryptographically Secure Random Number Generators (CSPRNGs). Our exploration of available literature has underscored the existing challenges in achieving absolute PRNG safety. We have determined that PRNG can only exist if $P \neq NP$, it can be constructed from one-way functions, and the output of the random number generator can be greatly expanded.

In the modern era with nowadays demand of secure communication channels, the indispensability of a robust supply of random or indistinguishable-from-random numbers is self-evident. All the most reliable protocols of password exchange that rely, among other things, on the random nonce. Practical applications have enthusiastically embraced algorithms designed to generate such numbers. Through the analysis of these algorithms, along with an examination of their historical evolution and vulnerabilities, certain conclusions emerge: the design of any RNG mechanism must meticulously account for potential attacks; systematic testing against all types of attacks with a proactive consideration of their possible consequences is crucial.

The most plausible types of attacks are state compromise leading to future simulation or backtracking, Chosen-Input Attacks. To secure PRNG against threats, design of such algorithm must possess the ability to rapidly recover after invasions and unintended exposure of internal states. Moreover, it should be made impossible to see the direct results of PRNG calculations (for example, use a one-way function before returning data). Secure PRNG must include an entropy accumulator, enabling a catastrophic reseed using new data that are independent of prior states and outputs. Regularly resetting the state, with a new value that is challenging to predict even in the presence of previous outputs, serves as an additional layer of defense.

As cryptographic systems continue to evolve, it is imperative to recognize that the landscape of security is dynamic. Future developments may necessitate further refinements in PRNG design and defense mechanisms. This study provides a foundation for such strives, emphasizing the importance of a proactive and adaptable approach to PRNG design, with an awareness of potential vulnerabilities and a commitment to robust countermeasures against emerging threats.

This final point highlights that keeping information safe is always dynamic. It reminds us that we must keep working hard to protect against new dangers in the world of computer security, which keeps changing. This allows us to keep up with the fast phased technological innovations thus creating a strong foothold into security protection.

Bibliography

[1] Menezes, Alfred J., van Oorschot, Paul C., Vanstone, Scott A. Handbook of Applied Cryptography CRC Press, 1996. ISBN: 0-8493-8523-7

[2] Luby, Michael. *Pseudorandomness and cryptographic applications*. Princeton computer science notes, Princeton University Press, 1996. ISBN 80-7378-001-1.

[3] Lehmer, Derrick H. *Mathematical methods in large-scale computing units*. In: Proceedings of 2nd Symposium on Large-Scale Digital Calculating Machinery, 1951. pp. 141-146

[4] Brickell, E.F., Odlyzko, A.M. *Cryptanalysis: a survey of recent results*. At: Proceedings of the IEEE, 1988. DOI: 10.1109/5.4443.

[5] Yu, HB., Zheng,, QX., Liu,, YJ. et al. *An improved method for predicting truncated multiple recursive generators with unknown parameters*. At: Des. Codes Cryptogr. 91, 2023. pp. 1713–1736 DOI: 10.1007/s10623-022-01175-4

[6] Sun, HY., Zhu,, QX., Zheng,, QX. et al. *Predicting truncated multiple recursive generators with unknown parameters*. At: Des. Codes Cryptogr. 88, 2020. pp. 1083–1102 DOI: 10.1007/s10623-020-00729-8

[7] Joux, A., Stern,, J. *Lattice reduction: a toolbox for the cryptanalyst*. J. Cryptol. 88, 1998. pp. 161–185

[8] David, G.C., Erich, K. *On fast multiplication of polynomials over arbitrary algebras*. Acta Informatica 28, 1991. pp. 693–701

[9] Boyar, J. *Inferring sequences produced by a linear congruential generator missing low-order bits*. At: J. Cryptology 1(3), 1989. pp. 177–184 DOI: 10.1007/BF02252875

[10] Stern, J. *Secret linear congruential generators are not cryptographically secure*. At: Proceedings of the 28th Annual Symposium on Foundations of Computer Science–SFCS, 1987. pp. 421-426. DOI: 10.1109/SFCS.1987.51

[11] Frieze, A., Håstad, J., Kannan, R., Lagarias, J., Shamir, A. *Reconstructing Truncated Integer Variables Satisfying Linear Congruences*. At: SIAM J. Comput. 17, 1988. pp. 262-280. DOI: 10.1137/0217016.

[12] Lenstra, A., Lenstra, A., Lovász, L. *Factoring Polynomials with Rational Coefficients*. At: Mathematische Annalen. 261, 1982. pp. 515-534. DOI: 10.1007/BF01457454.

[13] Micciancio, D., Goldwasser, S. *Complexity of Lattice Problems. A Cryptographic Perspective.* 2002 ISBN: 978-0-7923-7688-0 DOI: 10.1007/978-1-4615-0897-7

[14] Håstad, Johan, Impagliazzo, Russell, Levin,, Leonid A. et al. *A Pseudorandom Generator from any One-way Function.* At: SIAM Journal on Computing, 1999. pp. 1364–1396 DOI: 10.1137/S0097539793244708

[15] Ferguson,, Niels, Schneier, Bruce. *Practical cryptography.* 2003 ISBN: 978-0471223573

[16] Goldreich, Oded *Foundations of cryptography I: Basic Tools.* Cambridge: Cambridge University Press, 2001. ISBN: 978-0-511-54689-1

[17] Barker, Elaine (NIST), Roginsky, Allen (NIST). *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths* 2011 DOI: 10.6028/NIST.SP.800-131Ar1 available from http://dx.doi.org/10.6028/NIST.SP.800-131Ar1

[18] Semerjian, Hratch G. - Acting Director, NIST. *Federal Register / Vol. 70, No. 96* 2005 available from http://csrc.nist.gov/publications/fips/05-9945-DES-Withdrawl.pdf

[19] Dorrendorf, Leo, Gutterman, Zvi, Pinkas,, Benny. *Cryptanalysis of the random number generator of the Windows operating system.* At: ACM Trans. Inf. Syst. Secur., 2009. pp. 1-32 DOI: 10.1145/1609956.1609966

[20] RSA Laboratories, a division of RSA Data Security, Inc. *Open source code: RSAREF2.0* [cit. 2024-03-26] available from https://ftp.funet.fi/pub/crypt/cryptography/asymmetric/rsa/

[21] Kelsey, John, Schneier, Bruce, Wagner,, David A. et al. *Cryptanalytic Attacks on Pseudorandom Number Generators.* At: In: Vaudenay, S. (eds) Fast Software Encryption. FSE. Lecture Notes in Computer Science, vol 1372.1998 pp. 168-188 DOI: 10.1007/3-540-69710-1_12 ISBN: 978-3-540-69710-7

[22] Kelsey, John, Schneier, Bruce, Ferguson,, Niels. *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator.* At: Heys, H., Adams, C. (eds) Selected Areas in Cryptography. SAC. Lecture Notes in Computer Science, vol 1758. 1999 DOI: 10.1007/3-540-46513-8_2 ISBN: 978-3-540-46513-3

[23] Ferguson,, Niels, Schneier, Bruce, Kohno,, Tadayoshi. *Cryptography Engineering: Design Principles and Practical Applications.* 2010 DOI: 10.5555/1841202 ISBN: 978-0-470-47424-2

[24] *Article on the web site. Apple Platform Security: Random number generation* available from support.apple.com/en-hk/guide/security

[25] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* 2nd Edition 1996 DOI: 10.5555/1841202 ISBN: 978-0-470-47424-2

[26] Schneier, Bruce. *Blog.* [cit. 2024-04-06] available from https://www.schneier.com/blog/archives/2005/02/sha1_broken.html

[27] Bernstein, Daniel. *ChaCha, a variant of Salsa20* [cit. 2024-04-05] available from https://cr.yp.to/chacha/chacha-20080128.pdf

[28] Blum, Lenore, Blum, Manuel, Shub, Michael. *A Simple Unpredictable Pseudo-Random Number Generator* At: SIAM Journal on Computing. 15 (2), 1986. pp. 364–383 DOI: 10.1137/0215025. ISSN: 0097-5397

[29] Kneusel, Robet T. *Random Numbers and Computers* Springer, 2018 ISBN 978-3-319-77697-2

[30] Jenkins, Robert J. *Isaac.* At: International Workshop on Fast Software Encryption, pp. 41–49. Springer, 1996.

[31] de Koning Gans, Gerhard, Hoepman, Jaap-Henk, Garcia, Flavio D. *A Practical Attack on the MIFARE Classic.* At: Smart Card Research and Advanced Applications, 2010. pp. 267-282 ISBN: 978-3-540-85893-5

[32] Anonymous blog, 2020. [cit. 2024-04-07] available from https://medium.com/@popdogsec/dual-ec-drbg-a-look-into-a-potential-backdoor-395796e24ee6

[33] Kopp, Henning. *attacking a random number generator*, 2020. [cit. 2024-04-07] available from https://web.archive.org/web/20230927191527/https://www.schutzwerk.co
a-rng/

[34] Nohl, Karsten, Evans, David. *Reverse-Engineering a Cryptographic RFID Tag.* [cit. 2024-04-05] available from https://www.usenix.org/legacy/events/sec08/tech/full_papers/nohl/noh

[35] J. Jenkins Jr. , Robert. *ISAAC.* At: Gollmann, D. (eds) Fast Software Encryption. FSE 1996. pp. 41–49 ISBN: 978-3-540-49652-6 available from https://link.springer.com/content/pdf/10.1007/3-540-60865-6_41.pdf

[36] Pudovkina, Marina. *A known plaintext attack on the ISAAC keystream generator.* 2001 At: IACR Cryptology ePrint Archive. 2001. 49.

[37] U.S. Department of Commerce. *Digital Signature Standard (DSS)*. At: NIST FIPS PUB 186, 1994.

[38] Barker, E. *Digital Signature Standard (DSS) [includes Change Notice 1 from 12/30/1996]*. At: Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 1994.

[39] Accredited Standards Committee on Financial Services, X9, operating under the procedures of the American National Standards Institute. *Financial Institution Key Management (Wholesale)*. 1985.

[40] American National Standards Institute. *ANSI X3.92*. 1981.

[41] Krawczyk, Hugo. *How to predict congruential generators* At: J. Algorithms, 13. 1992. pp. 527-545 DOI: 10.1016/0196-6774(92)90054-G

[42] Boyar, J. *Inferring a sequence generated by a linear congruence* At: 23rd Annual Symposium on Foundations of Computer Science, 1982. pp. 153-159 DOI: 10.1109/SFCS.1982.73.

[43] Ajtai, M., Kumar, R., Sivakumar, D. *A sieve algorithm for the shortest lattice vector problem*. At: Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing. Association for Computing Machinery, New York, NY, USA. 2001. pp. 601–610

[44] Lenstra, A.K., Lenstra, H.W., Lovász, L. *Factoring polynomials with rational coefficients*. Math. Ann. 261(4), 1982. pp. 515–534

[45] Schnorr, C.P., Euchner, M. *Lattice basis reduction: improved practical algorithms and solving subset sum problems*. Math. Program. 66(2), 1994. pp. 181–199

A. Mersenne Twister attack. Source code

```
from z3 import *

def untemper(out):
    y1 = BitVec('y1', 32)
    y2 = BitVec('y2', 32)
    y3 = BitVec('y3', 32)
    y4 = BitVec('y4', 32)
    y = BitVecVal(out, 32)
    s = Solver()
    equations = [
        y2 == y1 ^ (LShR(y1, 11)),
        y3 == y2 ^ ((y2 << 7) & 0x9D2C5680),
        y4 == y3 ^ ((y3 << 15) & 0xEFC60000),
        y == y4 ^ (LShR(y4, 18))
    ]
    s.add(equations)
    s.check()
    return s.model()[y1].as_long()


def recover_state_mt(numbers):
    """
    This function recovers the internal state of MT19937 given a
    sequence of outputs. Note that there can be multiple states
    of an MT19937 that yield the same sequence of outputs.
    """
    state = []
    for n in numbers[0:624]:
        state.append(untemper(n))
    return state


def main():
    """
    This function tests the implementation.
    We clone the RNG from its output and compare the next
    generated outputs of the real and the cloned PRNG. Then,
    we try to recover the seed.
    """
    import random
    random.seed()
    a = random.getrandbits(32)
    print(f"seed esed: {a}")

    random.seed(a)
    random_nums = [random.getrandbits(32)]
    print("real internal state of PRNG:",
          f"{random.getstate()[1][0:10]} ...")
    print("generating random numbers")
    for i in range(623):
        random_nums.append(random.getrandbits(32))

    print(f"generated numbers: {random_nums[0:10]} ... ")
    print("recover internal state of PRNG")
    recovered_state = recover_state_mt(random_nums)
    print(f"recovered internal state: {recovered_state[0:10]}")

    print("continue generting with original sequence")
    real_seq = []
    for i in range(1000):
        real_seq.append(random.getrandbits(32))

    print("cloning PRNG")
    random.setstate((3,tuple(recovered_state+[624]),None))
    copy_seq = []
    for i in range(1000):
        copy_seq.append(random.getrandbits(32))

    print("checking equality of next 1000 outputs" +
          " from the real and cloned rng")
    for i in range(1000):
        assert(real_seq[i] == copy_seq[i])
    print('Success!')


main()
```