Pseudo Random Number Generation

Three Cases Where PRNGs Broke The System

>_ DEV v1.3-RC1

Ernesto Martínez García[1]
me@ecomaikgolf.com

Simon Lammer[2]
simon.lammer@student.tugraz.at

Graz University of Technology

Cryptanalysis VO SS23

22nd of June 2023

SLIDES & REPORT

ls.ecomaikgolf.com/slides/randomnumbers/

Motivation

Why Random Number Generation

- Importance might be forgotten, we usually depend on them.
- We try to break the mode or the primitive, but not the RNG.
- Bad RNGs can take down cryptosystems.

Objectives

- We wanted to show real world cases where RNGs broke the system
- For each case, explain the inner workings of the RNG and how they failed
- Plus a very special RNG ☺

Motivation

Why Random Number Generation

- Importance might be forgotten, we usually depend on them.
- We try to break the mode or the primitive, but not the RNG.
- Bad RNGs can take down cryptosystems.

Objectives

- We wanted to show real world cases where RNGs broke the system
- For each case, explain the inner workings of the RNG and how they failed
- Plus a very special RNG ☺

Motivation

Why Random Number Generation

- Importance might be forgotten, we usually depend on them.
- We try to break the mode or the primitive, but not the RNG.
- Bad RNGs can take down cryptosystems.

Objectives

- We wanted to show real world cases where RNGs broke the system
- For each case, explain the inner workings of the RNG and how they failed
- Plus a very special RNG ☺

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Playstation 3 Nonce Misuse

- Naivest Case of bad RNG
- Ended up in a PS3 Jailbreak

A Novel Related Nonce Attack for ECDSA

- Very Recent Attack
- 9.400.000 Dollars Affected

Dual Elliptic Curve Deterministic Random Bit Generator

- Standarized by NIST, ANSI, ISO for 7^{+} Years
- A Very Special Generator....

Elliptic Curve Essentials

- Which direction of computation is easy? (for known G)

$$k \times G \rightleftharpoons P$$

$$k \in \mathbb{N}, \quad G, P \in \mathrm{EC}$$

Elliptic Curve Essentials

- Which direction of computation is easy? (for known G)

$$k \times G \underset{\text{hard}}{\overset{\text{easy}}{\rightleftharpoons}} P$$

$$k \in \mathbb{N}, \quad G, P \in \mathrm{EC}$$

Playstation 3
Nonce Misuse

Introduction

¶ Sony used Elliptic Curve Digital Signatures for signed PS3 software updates.

ECDSA Recap

An ECDSA signature (r, s) can be created from a message m and a private key d

We agree on:

- A Elliptic Curve EC
- A basis point G on EC
- Order n of G
- A hash function h

Algorithm:

$k \xleftarrow{\$} [1, n-1]$ Randomly choose from uniform distribution.

$R = kG = (x_R, y_R)$

$r = x_R \bmod n$ If $r = 0$ restart the algorithm.

$e = h(m)$

$s = k^{-1}(e + d \times r) \bmod n$ If $s = 0$ restart the algorithm.

Introduction

¶ Sony used Elliptic Curve Digital Signatures for signed PS3 software updates.

ECDSA Recap

An ECDSA signature (r, s) can be created from a message m and a private key d

We agree on:

- A Elliptic Curve EC
- A basis point G on EC
- Order n of G
- A hash function h

Algorithm:

$$k \xleftarrow{\$} [1, n-1]$$ Randomly choose from uniform distribution.

$$R = kG = (x_R, y_R)$$

$$r = x_R \bmod n$$ If $r = 0$ restart the algorithm.

$$e = h(m)$$

$$s = k^{-1}(e + d \times r) \bmod n$$ If $s = 0$ restart the algorithm.

Importance of Randomness

$$s = \frac{e + d \cdot r}{k} \mod n$$

- What if an attacker gets to know k?

Importance of Randomness

$$s = \frac{e + d \cdot r}{k} \mod n$$

- What if an attacker gets to know k?

Importance of Randomness

$$s = \frac{e + d \cdot r}{k} \mod n$$

- ❓ What if an attacker gets to know k?
- ❗ Private Key 🔑 Recovery!

$$s = \frac{e + d \cdot r}{k} \quad \longrightarrow \quad d = \frac{s \cdot k - e}{r} \mod n$$

What Went Wrong?

- ☹ Sony used the worst possible randomness

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

- ☹ Sony used the worst possible randomness

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`

- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

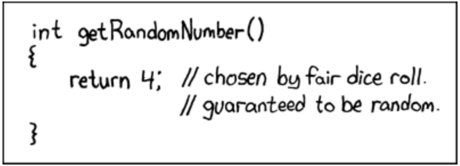
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot
  C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

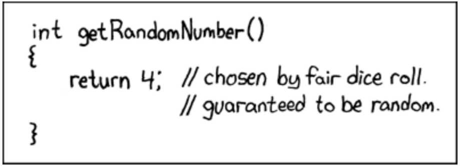
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot
  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

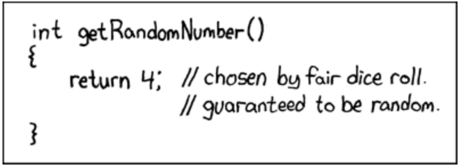
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

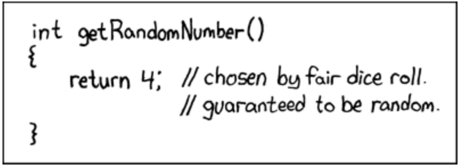
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

- ☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

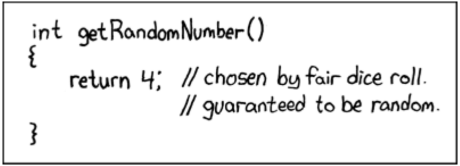
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`
- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

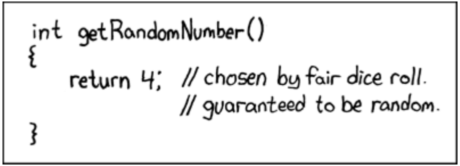
$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

What Went Wrong?

☹ Sony used the worst possible randomness: constant value k



Source: xkcd 221

- Discovered by group failOverflow (Dec. 2010)
- Signing keys got leaked by user geohot

  `C5B2 BFA1 A413 DD16 F26D 31C0 F2ED 4720 DCFB 0670`

- Jailbreaks for the PS3 were possible
- Couldn't be fixed for currently sold PS3

Key Recovery

$$\frac{s_1 \cdot k - e_1}{r_1} = d = \frac{s_2 \cdot k - e_2}{r_2}$$

$$k = \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1}$$

$$d = \frac{s_1}{r_1} \cdot \frac{e_1 \cdot r_2 - e_2 \cdot r_1}{s_1 \cdot r_2 - s_2 \cdot r_1} - \frac{e_1}{r_1} \mod n$$

A Novel Related Nonce Attack for ECDSA

Key observation

- Recall $d = \frac{k_i s_i - h(m_i)}{r_i} \mod n$.

$$\frac{k_0 s_0 - h_0}{r_0} = \frac{k_1 s_1 - h_1}{r_1} \implies k_1 = \frac{r_1 s_0}{r_0 s_1} k_0 + \frac{h_1 r_0 - h_0 r_1}{r_0 s_1} = u k_0 + v$$

- If these nonces obey a multivariate polynomial equation

$$a_0 k_0^{e_0} + a_1 k_1^{e_1} + a_2 k_2^{e_2} + \cdots + a_N = 0$$

- Furthermore, if a_i and e_i are known, the only unknown variable is d

$$a_0 \left(\frac{h_0}{s_0} + \frac{r_0}{s_0} d \right)^{e_0} + a_1 \left(\frac{h_1}{s_1} + \frac{r_1}{s_1} d \right)^{e_1} + a_2 \left(\frac{h_2}{s_2} + \frac{r_2}{s_2} d \right)^{e_2} + \cdots + a_N = 0$$

- The private key d appears in this polynomial's roots.

Key observation

- Recall $d = \frac{k_i s_i - h(m_i)}{r_i} \mod n$.

$$\frac{k_0 s_0 - h_0}{r_0} = \frac{k_1 s_1 - h_1}{r_1} \implies k_1 = \frac{r_1 s_0}{r_0 s_1} k_0 + \frac{h_1 r_0 - h_0 r_1}{r_0 s_1} = u k_0 + v$$

- If these nonces obey a multivariate polynomial equation

$$a_0 k_0^{e_0} + a_1 k_1^{e_1} + a_2 k_2^{e_2} + \cdots + a_N = 0$$

- Furthermore, if a_i and e_i are known, the only unknown variable is d

$$a_0 \left(\frac{h_0}{s_0} + \frac{r_0}{s_0} d\right)^{e_0} + a_1 \left(\frac{h_1}{s_1} + \frac{r_1}{s_1} d\right)^{e_1} + a_2 \left(\frac{h_2}{s_2} + \frac{r_2}{s_2} d\right)^{e_2} + \cdots + a_N = 0$$

- The private key d appears in this polynomial's roots.

Key observation

- Recall $d = \frac{k_i s_i - h(m_i)}{r_i} \mod n$.

$$\frac{k_0 s_0 - h_0}{r_0} = \frac{k_1 s_1 - h_1}{r_1} \implies k_1 = \frac{r_1 s_0}{r_0 s_1} k_0 + \frac{h_1 r_0 - h_0 r_1}{r_0 s_1} = u k_0 + v$$

- If these nonces obey a multivariate polynomial equation

$$a_0 k_0^{e_0} + a_1 k_1^{e_1} + a_2 k_2^{e_2} + \cdots + a_N = 0$$

- Furthermore, if a_i and e_i are known, the only unknown variable is d

$$a_0 \left(\frac{h_0}{s_0} + \frac{r_0}{s_0} d \right)^{e_0} + a_1 \left(\frac{h_1}{s_1} + \frac{r_1}{s_1} d \right)^{e_1} + a_2 \left(\frac{h_2}{s_2} + \frac{r_2}{s_2} d \right)^{e_2} + \cdots + a_N = 0$$

- The private key d appears in this polynomial's roots.

Key observation

- Recall $d = \frac{k_i s_i - h(m_i)}{r_i} \mod n$.

$$\frac{k_0 s_0 - h_0}{r_0} = \frac{k_1 s_1 - h_1}{r_1} \implies k_1 = \frac{r_1 s_0}{r_0 s_1} k_0 + \frac{h_1 r_0 - h_0 r_1}{r_0 s_1} = u k_0 + v$$

- If these nonces obey a multivariate polynomial equation

$$a_0 k_0^{e_0} + a_1 k_1^{e_1} + a_2 k_2^{e_2} + \cdots + a_N = 0$$

- Furthermore, if a_i and e_i are known, the only unknown variable is d

$$a_0 \left(\frac{h_0}{s_0} + \frac{r_0}{s_0} d \right)^{e_0} + a_1 \left(\frac{h_1}{s_1} + \frac{r_1}{s_1} d \right)^{e_1} + a_2 \left(\frac{h_2}{s_2} + \frac{r_2}{s_2} d \right)^{e_2} + \cdots + a_N = 0$$

- The private key d appears in this polynomial's roots.

Key observation

- Recall $d = \frac{k_i s_i - h(m_i)}{r_i} \mod n$.

$$\frac{k_0 s_0 - h_0}{r_0} = \frac{k_1 s_1 - h_1}{r_1} \implies k_1 = \frac{r_1 s_0}{r_0 s_1} k_0 + \frac{h_1 r_0 - h_0 r_1}{r_0 s_1} = u k_0 + v$$

- If these nonces obey a multivariate polynomial equation

$$a_0 k_0^{e_0} + a_1 k_1^{e_1} + a_2 k_2^{e_2} + \cdots + a_N = 0$$

- Furthermore, if a_i and e_i are known, the only unknown variable is d

$$a_0 \left(\frac{h_0}{s_0} + \frac{r_0}{s_0} d \right)^{e_0} + a_1 \left(\frac{h_1}{s_1} + \frac{r_1}{s_1} d \right)^{e_1} + a_2 \left(\frac{h_2}{s_2} + \frac{r_2}{s_2} d \right)^{e_2} + \cdots + a_N = 0$$

- The private key d appears in this polynomial's roots.

Recurrence relation PRNGs

- Attack works if PRNG used to generate nonces:
  - Uses arbitrary-degree recurrence relations modulo n ⟶ Only k_0 is truly random

$$k_1 = a_{N-3}k_0^{N-3} + a_{N-4}k_0^{N-4} + \cdots + a_1 k_0 + a_0$$

$$k_2 = a_{N-3}k_1^{N-3} + a_{N-4}k_1^{N-4} + \cdots + a_1 k_1 + a_0$$

$$k_3 = a_{N-3}k_2^{N-3} + a_{N-4}k_2^{N-4} + \cdots + a_1 k_2 + a_0$$

$$\vdots$$

$$k_{N-1} = a_{N-3}k_{N-2}^{N-3} + a_{N-4}k_{N-2}^{N-4} + \cdots + a_1 k_{N-2} + a_0$$

Goal

Produce a polynomial which only depends on the nonces, and not on unknown coefficients a_i

Recurrence relation PRNGs

- Attack works if PRNG used to generate nonces:
  - Uses arbitrary-degree recurrence relations modulo n → Only k_0 is truly random

$$k_1 = a_{N-3}k_0^{N-3} + a_{N-4}k_0^{N-4} + \cdots + a_1 k_0 + a_0$$

$$k_2 = a_{N-3}k_1^{N-3} + a_{N-4}k_1^{N-4} + \cdots + a_1 k_1 + a_0$$

$$k_3 = a_{N-3}k_2^{N-3} + a_{N-4}k_2^{N-4} + \cdots + a_1 k_2 + a_0$$

$$\vdots$$

$$k_{N-1} = a_{N-3}k_{N-2}^{N-3} + a_{N-4}k_{N-2}^{N-4} + \cdots + a_1 k_{N-2} + a_0$$

Goal

Produce a polynomial which only depends on the nonces, and not on unknown coefficients a_i

Recurrence relation PRNGs

- Attack works if PRNG used to generate nonces:
  - Uses arbitrary-degree recurrence relations modulo n → Only k_0 is truly random

$$k_1 = a_{N-3}k_0^{N-3} + a_{N-4}k_0^{N-4} + \cdots + a_1 k_0 + a_0$$

$$k_2 = a_{N-3}k_1^{N-3} + a_{N-4}k_1^{N-4} + \cdots + a_1 k_1 + a_0$$

$$k_3 = a_{N-3}k_2^{N-3} + a_{N-4}k_2^{N-4} + \cdots + a_1 k_2 + a_0$$

$$\vdots$$

$$k_{N-1} = a_{N-3}k_{N-2}^{N-3} + a_{N-4}k_{N-2}^{N-4} + \cdots + a_1 k_{N-2} + a_0$$

Goal

Produce a polynomial which only depends on the nonces, and not on unknown coefficients a_i

Recurrence relation PRNGs

- Attack works if PRNG used to generate nonces:
  - ☑ Uses arbitrary-degree recurrence relations modulo n ➞ Only k_0 is truly random

$$k_1 = a_{N-3}k_0^{N-3} + a_{N-4}k_0^{N-4} + \cdots + a_1 k_0 + a_0$$

$$k_2 = a_{N-3}k_1^{N-3} + a_{N-4}k_1^{N-4} + \cdots + a_1 k_1 + a_0$$

$$k_3 = a_{N-3}k_2^{N-3} + a_{N-4}k_2^{N-4} + \cdots + a_1 k_2 + a_0$$

$$\vdots$$

$$k_{N-1} = a_{N-3}k_{N-2}^{N-3} + a_{N-4}k_{N-2}^{N-4} + \cdots + a_1 k_{N-2} + a_0$$

Goal

Produce a polynomial which only depends on the nonces, and not on unknown coefficients a_i

Recurrence relation PRNGs

- Attack works if PRNG used to generate nonces:
  - Uses arbitrary-degree recurrence relations modulo n ➞ Only k_0 is truly random

$$k_1 = a_{N-3}k_0^{N-3} + a_{N-4}k_0^{N-4} + \cdots + a_1 k_0 + a_0$$

$$k_2 = a_{N-3}k_1^{N-3} + a_{N-4}k_1^{N-4} + \cdots + a_1 k_1 + a_0$$

$$k_3 = a_{N-3}k_2^{N-3} + a_{N-4}k_2^{N-4} + \cdots + a_1 k_2 + a_0$$

$$\vdots$$

$$k_{N-1} = a_{N-3}k_{N-2}^{N-3} + a_{N-4}k_{N-2}^{N-4} + \cdots + a_1 k_{N-2} + a_0$$

Goal

Produce a polynomial which only depends on the nonces, and not on unknown coefficients a_i

Example with Linear Congruential Generator PRNG

$$k_0 \xleftarrow{\$} [1, n-1]$$

$$k_1 = a_1 k_0 + a_0$$

$$k_2 = a_1 k_1 + a_0$$

$$k_3 = a_1 k_2 + a_0$$

$$k_1 - k_2 = a_1(k_0 - k_1)$$

$$a_1 = \frac{k_1 - k_2}{k_0 - k_1}$$

$$k_2 - k_3 = a_1(k_1 - k_2)$$

$$a_1 = \frac{k_2 - k_3}{k_1 - k_2}$$

$$(k_1 - k_2)^2 - (k_2 - k_3)(k_0 - k_1) = 0 \Longleftarrow \frac{k_1 - k_2}{k_0 - k_1} = \frac{k_2 - k_3}{k_1 - k_2}$$

Example with Linear Congruential Generator PRNG

$$k_0 \xleftarrow{\$} [1, n-1]$$

$$k_1 = a_1 k_0 + a_0$$

$$k_2 = a_1 k_1 + a_0$$

$$k_3 = a_1 k_2 + a_0$$

$$k_1 - k_2 = a_1(k_0 - k_1)$$

$$a_1 = \frac{k_1 - k_2}{k_0 - k_1}$$

$$k_2 - k_3 = a_1(k_1 - k_2)$$

$$a_1 = \frac{k_2 - k_3}{k_1 - k_2}$$

$$(k_1 - k_2)^2 - (k_2 - k_3)(k_0 - k_1) = 0 \Longleftarrow \frac{k_1 - k_2}{k_0 - k_1} = \frac{k_2 - k_3}{k_1 - k_2}$$

Example with Linear Congruential Generator PRNG

$$k_0 \xleftarrow{\$} [1, n-1]$$

$$k_1 = a_1 k_0 + a_0$$

$$k_2 = a_1 k_1 + a_0$$

$$k_3 = a_1 k_2 + a_0$$

$$k_1 - k_2 = a_1(k_0 - k_1)$$

$$a_1 = \frac{k_1 - k_2}{k_0 - k_1}$$

$$k_2 - k_3 = a_1(k_1 - k_2)$$

$$a_1 = \frac{k_2 - k_3}{k_1 - k_2}$$

$$(k_1 - k_2)^2 - (k_2 - k_3)(k_0 - k_1) = 0 \Longleftarrow \frac{k_1 - k_2}{k_0 - k_1} = \frac{k_2 - k_3}{k_1 - k_2}$$

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Impact

- Private keys from vulnerable signature sets can be found quickly.
  - Under 1s for a small number of related nonces N
  - ~ 6.5 s for $N = 16$, which yields a 92-degree polynomial
- The Bitcoin blockchain was tested (for N=5)
  - 424 million unique public keys
  - 9.1 million unique public keys with at least 5 signatures
  - 762 unique bitcoin wallets broken!
  - All of them reused nonces and had zero balance. ☹
  - Before they were exploited, these wallets contained about 144 BTC ($\sim$ 9.4M USD)
- Ethereum blockchain was also tested
  - No practical success
- Many unexplored applications remain, since ECDSA is widely used.

Dual Elliptic Curve Deterministic Random Bit Generator

🕵️

Introduction

¶ `DUAL_EC_DRBG` was a cryptographically secure deterministic random bit generator

History

- Developed by the NSA along others such as `HASH_DRBG`
- Originally standarized by ANSI, NIST and ISO followed
- Available in NIST's SP 800-90A `(10.6028/NIST.SP.800-90Ar1)`
- Deprecated from SP 800-90A in 2014 (from 2006)

Characteristics

- Makes use of Elliptic Curve Cryptography (Cryptography VO L8)
- Uses two Elliptic Curve points, that's where the "Double" come from
- Security is based on the Discrete Log EC Problem ($\mathbf{P} \cdot \mathbf{k} = \mathbf{Q}$)

Introduction

¶ `DUAL_EC_DRBG` was a cryptographically secure deterministic random bit generator

History

- Developed by the NSA along others such as `HASH_DRBG`
- Originally standarized by ANSI, NIST and ISO followed
- Available in NIST's SP 800-90A `(10.6028/NIST.SP.800-90Ar1)`
- Deprecated from SP 800-90A in 2014 (from 2006)

Characteristics

- Makes use of Elliptic Curve Cryptography (Cryptography VO L8)
- Uses two Elliptic Curve points, that's where the "Double" come from
- Security is based on the Discrete Log EC Problem ($\mathbf{P} \cdot \mathbf{k} = \mathbf{Q}$)

Introduction

¶ `DUAL_EC_DRBG` was a cryptographically secure deterministic random bit generator

History

- Developed by the NSA along others such as `HASH_DRBG`
- Originally standarized by ANSI, NIST and ISO followed
- Available in NIST's SP 800-90A `(10.6028/NIST.SP.800-90Ar1)`
- Deprecated from SP 800-90A in 2014 (from 2006)

Characteristics

- Makes use of Elliptic Curve Cryptography (Cryptography VO L8)
- Uses two Elliptic Curve points, that's where the "Double" come from
- Security is based on the Discrete Log EC Problem ($\mathbf{P} \cdot \mathbf{k} = \mathbf{Q}$)

Algorithm I

Parameters

- E: $y^2 = x^3$ - 3x + 0x5a...4b mod 11...51
- n: 1157...4369
- $P \in E$: (0x6b...96, 0x4f...f5)
- $Q \in E$: (0xc9...92, 0xb2...46)

Operations

- Seed: S_0
- f(): $S_i \cdot P$ (+ more)
- g(): $S_i \cdot Q$ (+ more)
- Out: r_i

¶ Keeps an inner state **(red)** and an outer state **(green)**

Algorithm I

Parameters

- E: $y^2 = x^3$ - 3x + 0x5a...4b mod 11...51
- n: 1157...4369
- $P \in E$: (0x6b...96, 0x4f...f5)
- $Q \in E$: (0xc9...92, 0xb2...46)

Operations

- Seed: S_0
- f(): $S_i \cdot P$ (+ more)
- g(): $S_i \cdot Q$ (+ more)
- Out: r_i

¶ Keeps an inner state **(red)** and an outer state **(green)**

Algorithm I

Parameters

- E: $y^2 = x^3$ - 3x + 0x5a...4b mod 11...51
- n: 1157...4369
- $P \in E$: (0x6b...96, 0x4f...f5)
- $Q \in E$: (0xc9...92, 0xb2...46)

Operations

- Seed: S_0
- f(): $S_i \cdot P$ (+ more)
- g(): $S_i \cdot Q$ (+ more)
- Out: r_i

¶ Keeps an inner state **(red)** and an outer state **(green)**

Algorithm I

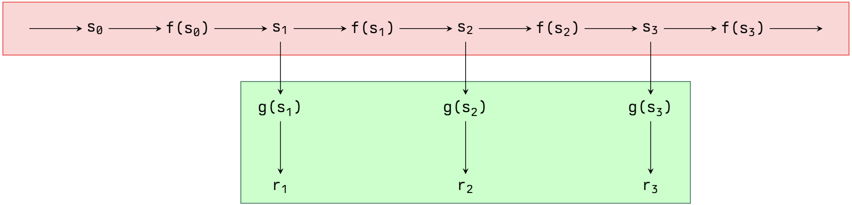
Parameters

- E: $y^2 = x^3$ - 3x + 0x5a...4b mod 11...51
- n: 1157...4369
- $P \in E$: (0x6b...96, 0x4f...f5)
- $Q \in E$: (0xc9...92, 0xb2...46)

% Operations

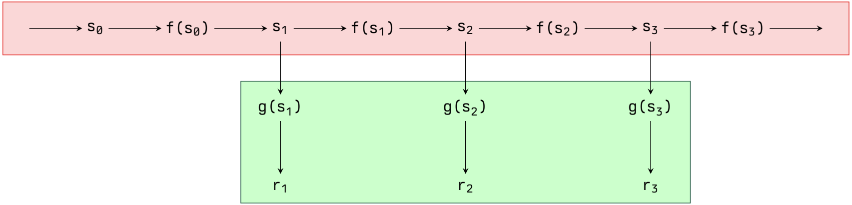
- Seed: S_0
- f(): $S_i \cdot P$ (+ more)
- g(): $S_i \cdot Q$ (+ more)
- Out: r_i

¶ Keeps an inner state **(red)** and an outer state **(green)**



Algorithm II

¶ Keeps an inner state **(red)** and an outer state **(green)**



Inner state is protected by ECDLP

- We cannot, from a `(Q = kP)` point, recover `(P)` and obtain (s_i)

Seed recovery is protected by ECDLP

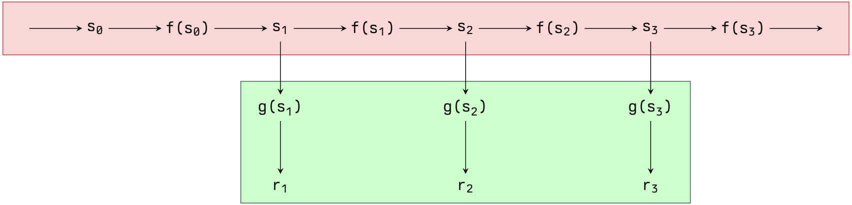
- We cannot, from a `(Q = kP)` point, recover `(P)` and move backwards obtaining (s_{i-1})

Having (s_i) means being able to compute $(s_{j > i})$

- Recovering the inner state is disasterous. An attacker can predict bits with `100%` accuracy

Algorithm II

¶ Keeps an inner state **(red)** and an outer state **(green)**



Inner state is protected by ECDLP

- We cannot, from a `(Q = kP)` point, recover `(P)` and obtain (s_i)

Seed recovery is protected by ECDLP

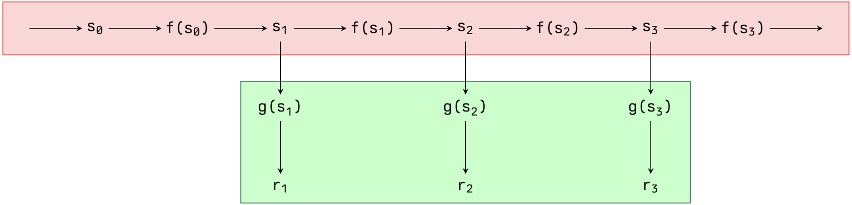
- We cannot, from a `(Q = kP)` point, recover `(P)` and move backwards obtaining (s_{i-1})

Having (s_i) means being able to compute $(s_{j>i})$

- Recovering the inner state is disasterous. An attacker can predict bits with `100%` accuracy

Algorithm II

¶ Keeps an inner state **(red)** and an outer state **(green)**



Inner state is protected by ECDLP

- We cannot, from a `(Q = kP)` point, recover `(P)` and obtain (s_i)

Seed recovery is protected by ECDLP

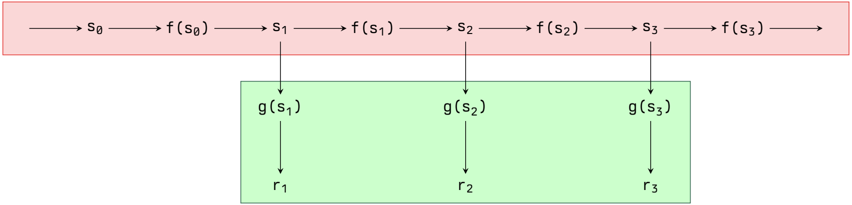
- We cannot, from a `(Q = kP)` point, recover `(P)` and move backwards obtaining (s_{i-1})

Having (s_i) means being able to compute $(s_{j>i})$

- Recovering the inner state is disasterous. An attacker can predict bits with `100%` accuracy

Algorithm II

¶ Keeps an inner state **(red)** and an outer state **(green)**



Inner state is protected by ECDLP

- We cannot, from a $(Q = kP)$ point, recover (P) and obtain (s_i)

Seed recovery is protected by ECDLP

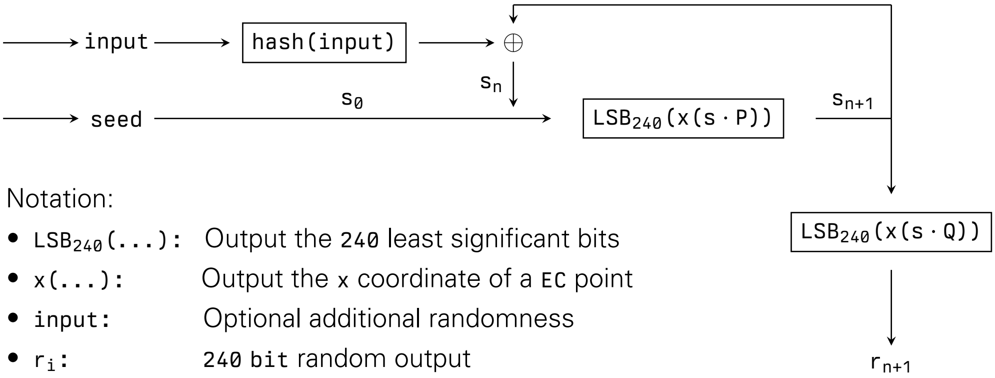
- We cannot, from a $(Q = kP)$ point, recover (P) and move backwards obtaining (s_{i-1})

Having (s_i) means being able to compute $(s_{j>i})$

- Recovering the inner state is disasterous. An attacker can predict bits with 100% accuracy

Algorithm III

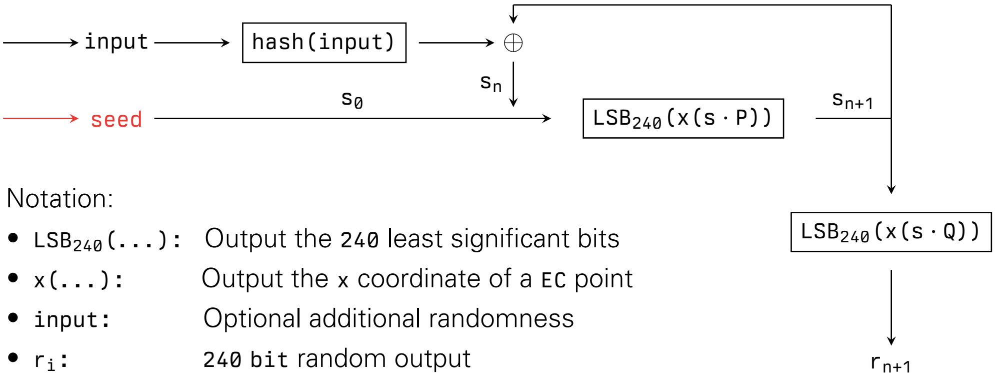
- How the Algorithm Really Works:



- Notation:
  - `LSB240(...)`: Output the `240` least significant bits
  - `x(...)`: Output the `x` coordinate of a `EC` point
  - `input`: Optional additional randomness
  - r_i: `240 bit` random output
  - s_i: `256 bit` inner state
  - `seed`: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

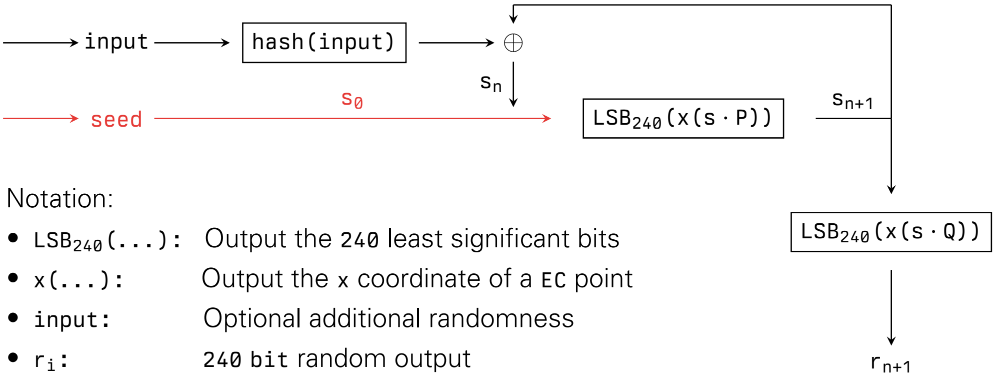


Notation:

- $LSB_{240}(...)$: Output the 240 least significant bits
- $x(...)$: Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Algorithm III

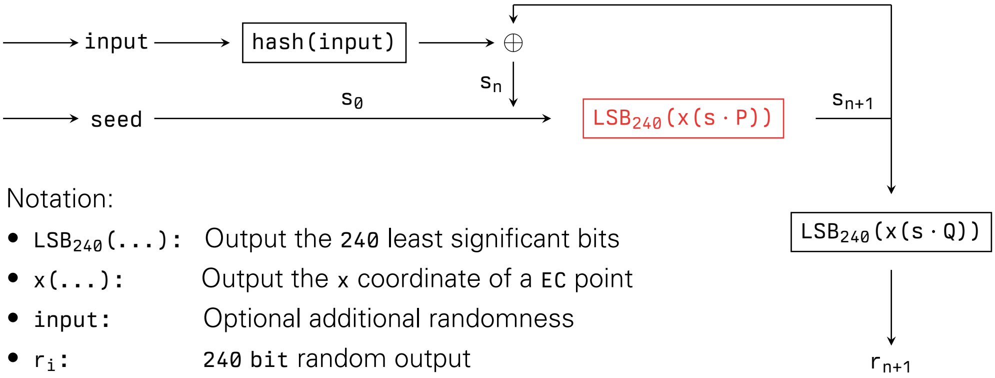
- How the Algorithm Really Works:



- Notation:
  - LSB$_{240}$(...): Output the 240 least significant bits
  - x(...): Output the x coordinate of a EC point
  - input: Optional additional randomness
  - r_i: 240 bit random output
  - s_i: 256 bit inner state
  - seed: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

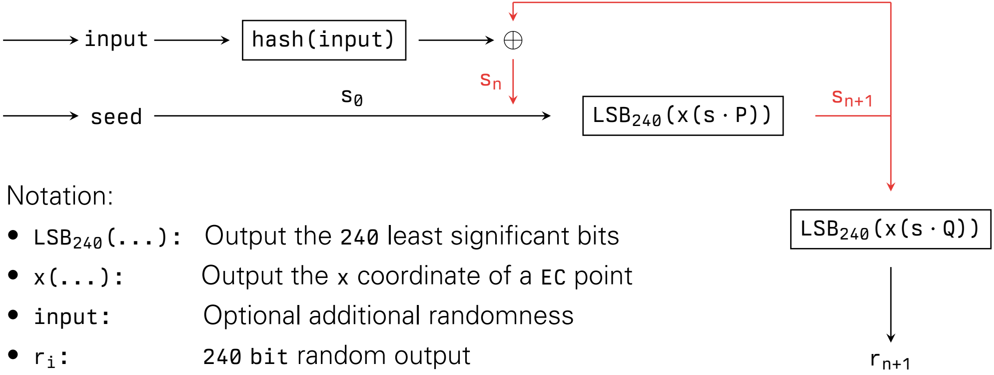


Notation:

- $\mathtt{LSB}_{240}$(...): Output the 240 least significant bits
- x(...): Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

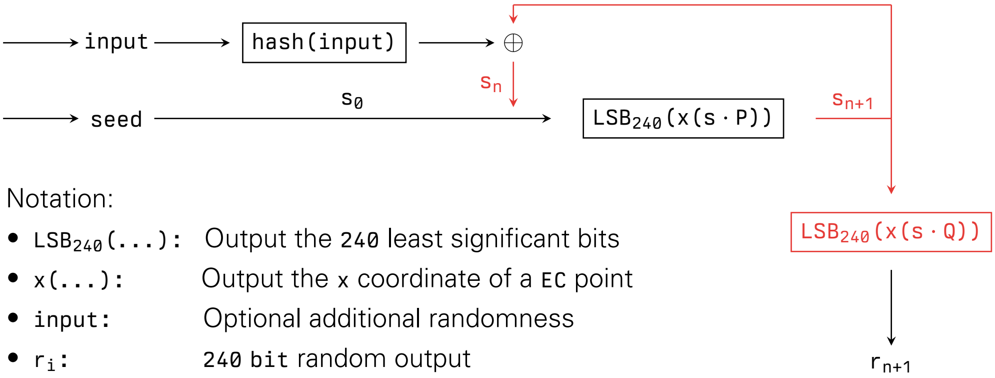


Notation:

- $LSB_{240}(...)$: Output the 240 least significant bits
- x(...): Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

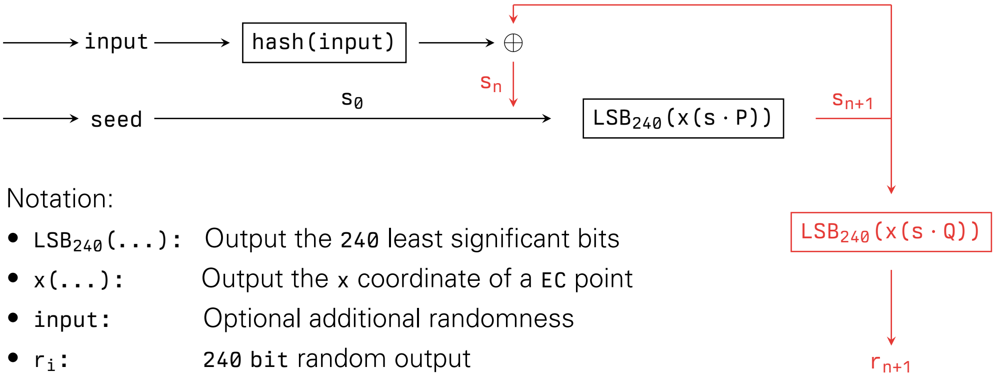


Notation:

- $\text{LSB}_{240}(\ldots)$: Output the 240 least significant bits
- $x(\ldots)$: Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

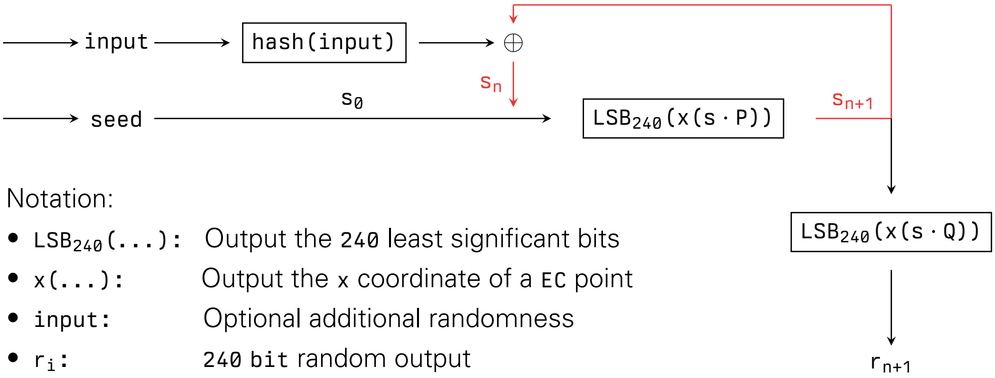


Notation:

- `LSB240(...)`: Output the `240` least significant bits
- `x(...)`: Output the `x` coordinate of a `EC` point
- `input`: Optional additional randomness
- r_i: `240 bit` random output
- s_i: `256 bit` inner state
- `seed`: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

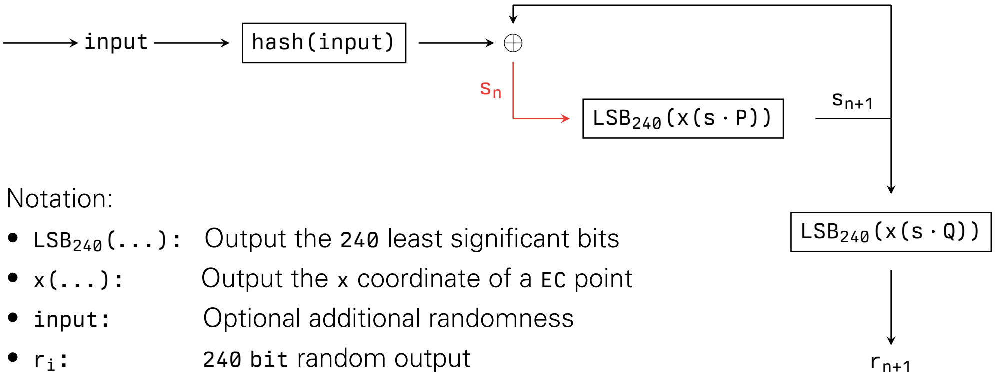


Notation:

- LSB_{240}(...): Output the 240 least significant bits
- x(...): Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Algorithm III

How the Algorithm Really Works:

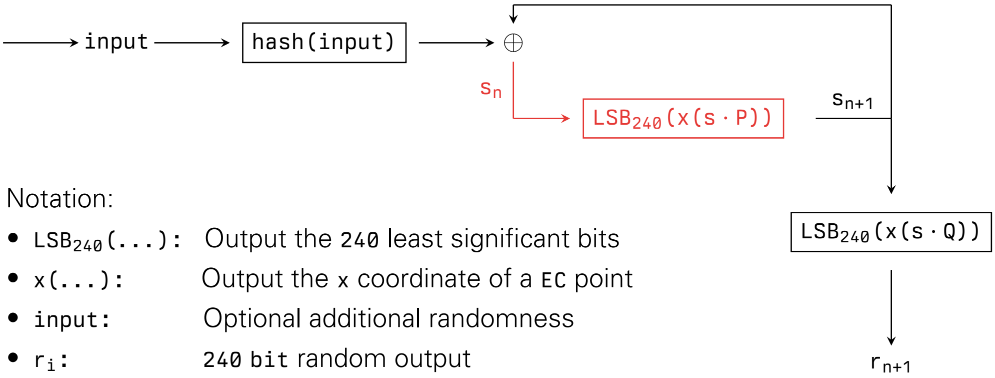


Notation:

- `LSB`$_{240}$`(...)`: Output the `240` least significant bits
- `x(...)`: Output the `x` coordinate of a `EC` point
- `input`: Optional additional randomness
- r_i: `240 bit` random output
- s_i: `256 bit` inner state
- `seed`: Initial source of randomness

Algorithm III

How the Algorithm Really Works:



Notation:

- $LSB_{240}(...)$: Output the 240 least significant bits
- $x(...)$: Output the x coordinate of a EC point
- input: Optional additional randomness
- r_i: 240 bit random output
- s_i: 256 bit inner state
- seed: Initial source of randomness

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points `(X,Y) = R = sQ` can be bruteforced

   ☹ But knowing the outer point `R = sQ = (X,Y)` point is not useful

   We might now know `R = sQ`, but we are interested on the `s` to calculate next states:

   $$s = LSB_{240}(x(s \cdot P))$$

   And that means breaking ECDLP `(R = sQ)`

💡 But Bob came with an amazing (and scary) idea.
What if Eve 🕵 knows a secret relation `e` between `P` and `Q`?

$$P = e \cdot Q$$

2. Eve calculates all possible `R = (X,Y)` from a r_i. As $(R = s \cdot Q)$ she multiplies it by `e` 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$e \cdot R = s \cdot P$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X, Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X, Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.
What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X, Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.
What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

- 🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced
   - ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

     We might now know $R = sQ$, but we are interested on the s to calculate next states:

     $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

     And that means breaking ECDLP $(R = sQ)$

- 💡 But Bob came with an amazing (and scary) idea.

  What if Eve 🕵 knows a secret relation e between P and Q?

  $$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

We might now know $R = sQ$, but we are interested on the s to calculate next states:

$$\boxed{s = LSB_{240}(x(s \cdot P))}$$

And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X, Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X, Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X, Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

$$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

$$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick I

🪄 Bob, scared of Eve 🕵 studied the algorithm and found some interesting properties

1. With a single (r_i) all possible 2^{16} curve points $(X,Y) = R = sQ$ can be bruteforced

   ☹ But knowing the outer point $R = sQ = (X,Y)$ point is not useful

   We might now know $R = sQ$, but we are interested on the s to calculate next states:

   $$\boxed{s = LSB_{240}(x(s \cdot P))}$$

   And that means breaking ECDLP $(R = sQ)$

💡 But Bob came with an amazing (and scary) idea.

What if Eve 🕵 knows a secret relation e between P and Q?

$$P = e \cdot Q$$

2. Eve calculates all possible $R = (X,Y)$ from a r_i. As $(R = s \cdot Q)$ she multiplies it by e 🪄!

$$e \cdot R = e \cdot s \cdot Q$$

$$e \cdot R = s \cdot e \cdot Q$$

$$\boxed{e \cdot R = s \cdot P}$$

Magic Trick II

- What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

❓ What did just happen?

Eve 🕵 created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \mod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

⚠ WHERE DO THE NIST PARAMETERS CAME FROM?!?!

🕵

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

❓ What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

⚠ WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters (P,Q). She fixed P and generated a scalar d:

$$d \cdot P = Q$$

Then found an e such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just 240 bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Magic Trick II

What did just happen?

Eve created backdoored public parameters `(P,Q)`. She fixed `P` and generated a scalar `d`:

$$d \cdot P = Q$$

Then found an `e` such that $e \cdot d = 1 \bmod r$:

$$e \cdot d \cdot P = e \cdot Q$$

$$P = e \cdot Q$$

With just `240` bits of random output, she can predict all the following bits.

But... this was standarized in NIST for 7 years, and used by default in crypto libraries.

WHERE DO THE NIST PARAMETERS CAME FROM?!?!

Tinfoil Hat I

- NSA got asked about it in a meeting. NSA's response:

Tinfoil Hat I

- NSA got asked about it in a meeting. NSA's response:

"NSA generated `(P,Q)` in a secure, classified way."

Tinfoil Hat II

- Community had the idea of randomly generating `Q`. NIST member response:

"`Q` [...] could also be generated [...], but NSA kiboshed this idea, and I was not allowed to publicly discuss it, just in case you may think of going there"

Tinfoil Hat II

- Community had the idea of randomly generating `Q`. NIST member response:

"`Q` [...] could also be generated [...], but NSA kiboshed this idea, and I was not allowed to publicly discuss it, just in case you may think of going there"

Tinfoil Hat III

- Why this was the default in RSA BSAFE crypto library? Reuters 📰 :

"... RSA received 10.000.000 dollars in a deal that set the NSA formula as the preferred, or default, method for number generation in the BSafe software, according to two sources familiar with the contract ... [10M] represented more than a third of the revenue that the relevant division at RSA had taken during the entire previous year..."

Tinfoil Hat III

- Why this was the default in RSA BSAFE crypto library? Reuters :

"... RSA received 10.000.000 dollars in a deal that set the NSA formula as the preferred, or default, method for number generation in the BSafe software, according to two sources familiar with the contract ... [10M] represented more than a third of the revenue that the relevant division at RSA had taken during the entire previous year..."

Tinfoil Hat IV

- 2013: Edward Snowden's NSA leaks are published.

NSA Bullrun program existence is revealed. Program's goal was to:

> "...covertly introduce weaknesses into the encryption standards..."

✖ Afterwards, NSA recommended stop using `DUAL_EC_DRBG`. Rest follow the advice.

Tinfoil Hat IV

- 2013: Edward Snowden's NSA leaks are published.

NSA Bullrun program existence is revealed. Program's goal was to:

"...covertly introduce weaknesses into the encryption standards..."

- ✖ Afterwards, NSA recommended stop using `DUAL_EC_DRBG`. Rest follow the advice.

Tinfoil Hat IV

- 2013: Edward Snowden's NSA leaks are published.

NSA Bullrun program existence is revealed. Program's goal was to:

"...covertly introduce weaknesses into the encryption standards..."

- Afterwards, NSA recommended stop using `DUAL_EC_DRBG`. Rest follow the advice.

Tinfoil Hat IV

- 2013: Edward Snowden's NSA leaks are published.

NSA Bullrun program existence is revealed. Program's goal was to:

"...covertly introduce weaknesses into the encryption standards..."

- Afterwards, NSA recommended stop using `DUAL_EC_DRBG`. Rest follow the advice.

Tinfoil Hat IV

- 2013: Edward Snowden's NSA leaks are published.

NSA Bullrun program existence is revealed. Program's goal was to:

> "...covertly introduce weaknesses into the encryption standards..."

- ✖ Afterwards, NSA recommended stop using `DUAL_EC_DRBG`. Rest follow the advice.

Backdoor Proof

Now let's mathematically prove the existence of the backdoor, so we can sue NSA

We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

But wait... that sounds familiar. Isn't this the ECDLP?

To prove the existence of a backdoor we would have to break ECDLP.

They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Backdoor Proof

- Now let's mathematically prove the existence of the backdoor, so we can sue NSA
- We have to prove that there is a relation between NIST's `P` and `Q`

1. By having `P` and `Q` we have to find one of the following numbers:

$P \cdot d = Q$	$P = e \cdot Q$

- But wait... that sounds familiar. Isn't this the ECDLP?
- To prove the existence of a backdoor we would have to break ECDLP.
- They used cryptography to hide them using cryptography to break cryptography *supposedly*

Ironic

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

- To conclude let’s add a few notes to the history

- NIST knew about the possible backdoor.
  - Argued that “there was no evidence of those numbers existing”

- `DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.
  - Thousands of times slower than alternatives
  - Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

- NIST added the possibility to generate your own parameters
  - In the Appendix, and you wouldn’t get FIPS validation.
  - Nobody generated their own values.

- NSA said that they wanted it in the standard so they could use it
  - Believed nobody would use it because “it’s ugly and slow”
  - Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

- To conclude let's add a few notes to the history

- NIST knew about the possible backdoor.
  - Argued that "there was no evidence of those numbers existing"

- `DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.
  - Thousands of times slower than alternatives
  - Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

- NIST added the possibility to generate your own parameters
  - In the Appendix, and you wouldn't get FIPS validation.
  - Nobody generated their own values.

- NSA said that they wanted it in the standard so they could use it
  - Believed nobody would use it because "it's ugly and slow"
  - Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Extra Notes

To conclude let's add a few notes to the history

NIST knew about the possible backdoor.

- Argued that "there was no evidence of those numbers existing"

`DUAL_EC_DRBG` was objectively worse than other alternatives. Still got standarized.

- Thousands of times slower than alternatives
- Output bias, guessing with sucess rate of `0.50078`. Unacceptable in all other cases.

NIST added the possibility to generate your own parameters

- In the Appendix, and you wouldn't get FIPS validation.
- Nobody generated their own values.

NSA said that they wanted it in the standard so they could use it

- Believed nobody would use it because "it's ugly and slow"
- Shared their `(P,Q)` in case anybody wanted to use them, but people could generate their own

Consequences: The Hash of Shame

Consequences: The Hash of Shame

mjos\dwez
@mjos_crypto

If NIST keeps line 2, SHA3-256 hash of the 256-bit random number generated on line 1, I'll just call it "the hash of shame."
It's there because the designers of Kyber think that RNGs (or NIST RBGs) are so bad that they need post-processing like this. You know, just in case.

Algorithm 8 $\textsc{Kyber}$.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$

1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \mathrm{H}(m)$
3: $(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
4: $c := \textsc{Kyber}.\mathsf{CPAPKE.Enc}(pk, m, r)$
5: $K := \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
6: **return** (c, K)

Question Time

☺

Pseudo Random Number Generation

Three Cases Where PRNGs Broke The System

>_ PROD v1.3 ✔

Ernesto Martínez García [1]
me@ecomaikgolf.com

Simon Lammer [2]
simon.lammer@student.tugraz.at

Graz University of Technology

Cryptanalysis VO SS23

22nd of June 2023

SLIDES & REPORT

ls.ecomaikgolf.com/slides/randomnumbers/