Chaotic sequences based image encryption

Kirill Trofimov
Institute of Radioelectronics
Riga Technical University
Riga, Latvia
kirils.trofimovs@edu.rtu.lv

Sergej Umnov
Institute of Radioelectronics
Riga Technical University
Riga, Latvia
sergejs.umnovs@edu.rtu.lv

***Abstract*— In this paper we propose an image encryption method based on chaotic sequences produced by Vilnius oscillator. The solution is provided as a VHDL model and is tested against implementation in Matlab.**

***Index terms*—vilnius oscillator, image encryption, chaos**

I. Introduction

Image encryption is a common process in modern communication and storage systems. It is used to protect confidentiality and integrity of digital images from unauthorized access. Considering the omnipresence of visual data, it's processing takes a considerable amount of compute resources. Chaos-based image encryption supposedly provides a higher data rate and compute efficiency than traditional image encryption methods. In this paper, a simple chaotic system is used[1] as a pseudo-random number generator. A bitstream produced by the generator is then used to diffuse the pixel data of the plain image, resulting in a cipher image. The result of this research is a matlab and VHDL model that implements both the Vilnius oscillator and the image encryption process.

II. Methods

Chaotic sequence that is required for encryption is produced using Vilnius oscillator.

A. Vilnius oscillator

Study depends on the Vilnius oscillator[1] as a chaos oscillator to generate PRNG sequences. Circuit diagram of the Vilnius oscillator is given in Figure 1.

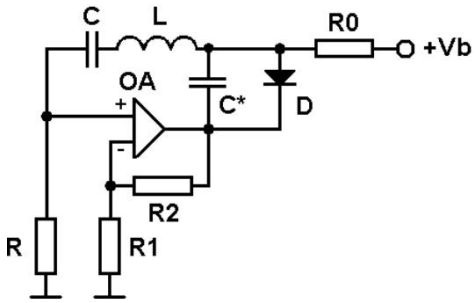

Figure 1: Vilnius oscillator circuit diagram [1]

A system of equations that defines the Vilnius oscillator is shown in (1) [1].

$$\begin{cases} C_1 \frac{\mathrm{d}V_{C_1}}{\mathrm{d}t} = I_L \\ L \frac{\mathrm{d}I_L}{\mathrm{d}t} = (k-1)RI_L - V_{C_1} - V_{C_2} \\ C_2 \frac{\mathrm{d}V_{C_2}}{\mathrm{d}t} = I_0 + I_L - I_D \end{cases} \tag{1}$$

Which is then presented in a more convinient form for simulation in (2) [1].

$$\begin{cases} \dot{x} = y \\ \dot{y} = ay - x - z \\ \varepsilon \dot{z} = b + y - c(\exp z - 1) \end{cases} \tag{2}$$

Constants used for simulation are following:

$R_1 = 1 \cdot 10^3 \Omega \quad R_2 = 10 \cdot 10^3 \Omega$
$R_3 = 6 \cdot 10^3 \Omega \quad R_4 = 20 \cdot 10^3 \Omega$
$C_1 = 1 \cdot 10^{-9} F \quad C_2 = 150 \cdot 10^{-12} F$
$L = 1 \cdot 10^{-3} H$

When oscillators function crosses $Y = 0$, related X value is compared to an arbitrarily chosen threshold as in formula (3).

$$\begin{cases} 1 \text{ if } x < 60 \wedge y = 0 \\ 0 \text{ if } x \geq 60 \wedge y = 0 \end{cases} \tag{3}$$

Bitstream generation method is shown in Figure 2.

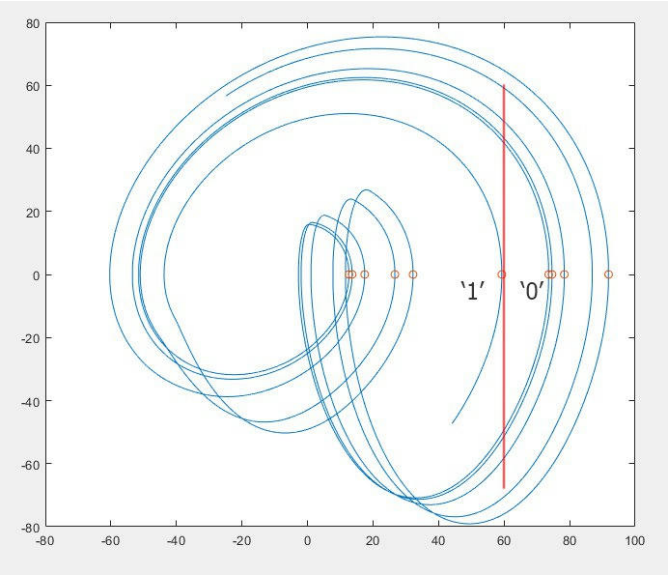

Figure 2: Bitstream generation method

B. *Encryption*

Cipher image is produced by XOR operation using generated pseudo-random bitstream and bits of the plain image as in (4), where IMG is the bitstream of the plain image and PRNG is the bitstream generated by the Vilnius oscillator.

$$C = \text{IMG} \oplus \text{PRNG} \tag{4}$$

Such encryption is implemented by doing XOR operation on batches of bits, since manipulating single bits is extremely inefficient.

C. *FPGA*

D. *Simulations*

While the main emphasis of this work is to make an FPGA model that implements the Vilnius oscillator and encryption, a Matlab model was made to make comparsion against it, and also because FPGA model simulation is an extremely resource-consuming process. Simulation source codes and execution instructions are available in a github repository[1].

1) *Matlab simulation:*

Encryption is done by going through each $Y = 0$ crossing, which are obtained by solving (2) differential equation, collecting 8 bits into a byte and doing XOR operation with corresponding byte of the plain image. Following is the code of encryption in matlab simulation:

```
[t, var] = ode45(
  @(t, var) calc_derivatives(t, var, N),
  [min_time, max_time],
  x0, ode_options
);

y = var(:,2);
x = var(:,1);

y_cross_indices = find(diff(sign(y)) < 0);
y_cross_indices = y_cross_indices(2:end);

bits = x(y_cross_indices) < 60;
if length(bits) >= 8
  bits = 1*(bits(1:8))';
else
  bits = 1*[bits' zeros(1, 8-length(bits))];
end

img_bits = de2bi(img(i, j),8);
img_encr(i, j) = bi2de(xor(bits, img_bits));
```

Since in FPGA environment exponent approximation is required, it is also simulated in matlab.

2) *FPGA simulation:*

Simulation in an FPGA environment is done using Intel Questa software. In Figure 3 we are comparing attractor field diagrams of Vilnius oscillators simulated with the FPGA model and Matlab model.

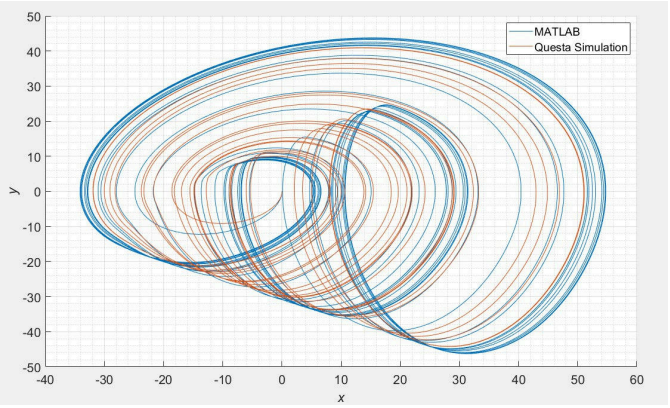

Figure 3: FPGA simulation compared to Matlab simulation

E. *Exponent approximation*

The need to approximate the exponent function on FPGA arises from the fact that the exponential function is computationally expensive and requires a large number of resources to compute accurately. Following is the representation of exponent approximation in matlab:

```
function value = exp_approx(x, N, x_min, x_max)
  x_step = (x_max-x_min)/N;
  value = 0;
  for i = 0:N-1
    x1 = x_min + i*x_step;
    x2 = x1 + x_step;
    if x >= x1
      y1 = exp(x1);
      y2 = exp(x2);
      value = (x-x1)/(x2-x1)*(y2-y1) + y1;
    end
  end
end
```

[1]https://github.com/entritarus/hw_chaos

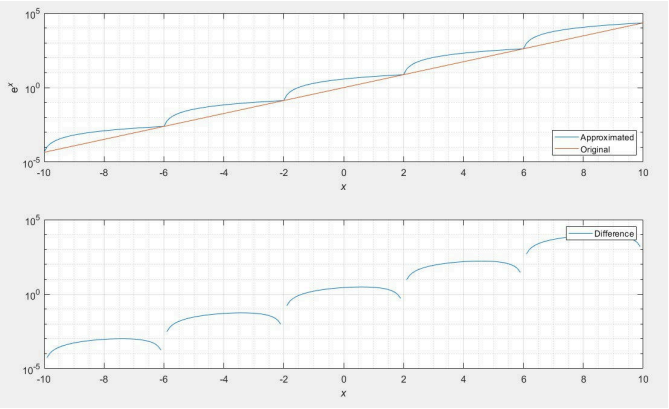

Figure 4: Difference between approximated exponent and e^x

III. Results

While generally the source images are encrypted, nonuniform distribution is clearly visible, especially in Figure 5, where the dark region has impact on the same region in the resulting image. Nonuniform distribution of chipher data is confirmed by comparing histograms of images before and after encryption in Figure 6, Figure 8 and Figure 10. Ideally, every byte of encrypted data should appear as frequently as any other byte, which our encryption fails to achieve.

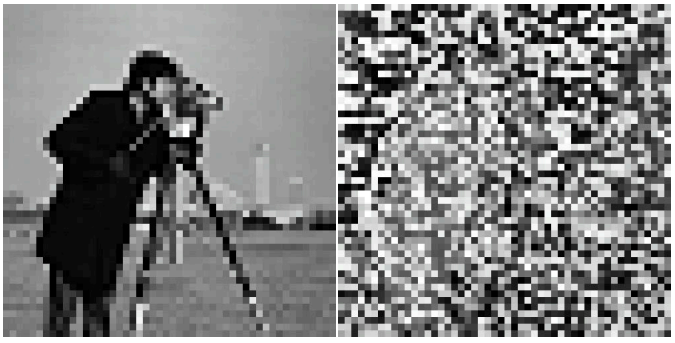

Figure 5: Cameraman: plain image, cipher image

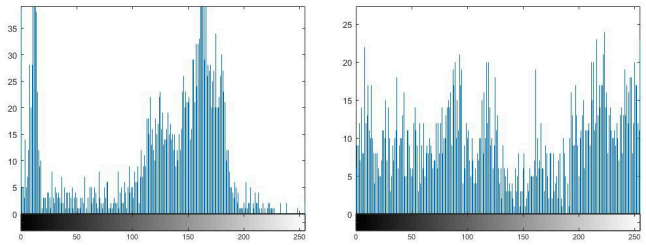

Figure 6: Cameraman: plain image, cipher image histograms

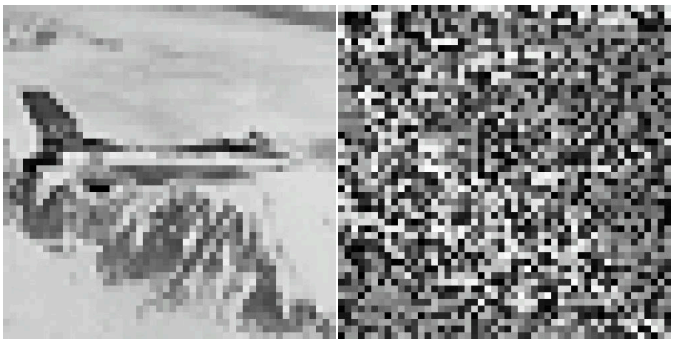

Figure 7: Jetplane: plain image, cipher image

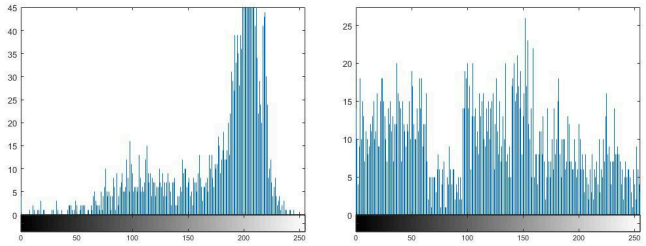

Figure 8: Jetplane: plain image, cipher image histograms

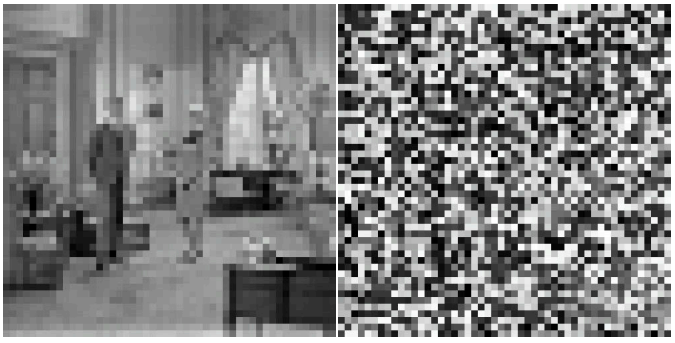

Figure 9: Livingroom: plain image, cipher image

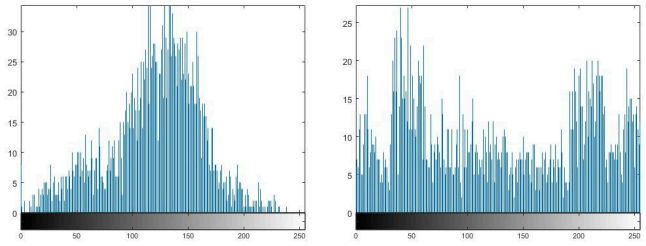

Figure 10: Livingroom: plain image, cipher image histograms

Two additional metrics were calculated for each sample: the number of changing pixel rate (NPCR) and the unified averaged changed intensity (UACI) [2].

Image	NPCR	UACI
Cameraman	100%	21.61%
Jetplane	100%	6.62%
Livingroom	100%	17.31%

Table 1: NPCR and UACI results for encryption

IV. Conclusion

In this paper we managed to achieve encryption of images using chaos sequence produced by Vilnius oscillator. Implementation of encryption and chaos sequence generation is done both as an FPGA model and a Matlab model.

Simple diffusion using XOR operation seems to be lacking in encryption quality. This is because plain image has high impact on the result of XOR operation, by shifting probability of pixel values according to pixel position. It is possible to improve encryption quality by shuffling the pixels using the same chaos sequence as an additional step. Repeating diffusion and shuffle steps multiple times might lead to better encryption quality.

References

[1] A. Tamasevicius, G. Mykolaitis, K. Pyragas, and V. Pyragas, "A simple chaotic oscillator for educational purposes", *European Journal of Physics*, vol. 26, pp. 61–63, 2004, doi: 10.1088/0143-0807/26/1/007.

[2] Y. Wu, "NPCR and UACI Randomness Tests for Image Encryption", *Cyber Journals: Journal of Selected Areas in Telecommunications*, p., 2011.