FPGA Based Compressive Sensing and Chaotic Encryption for Secure Image Transmission

PROJECT-21ECP302L

Submitted by

Abhinav Kumar [Reg No:RA2211004010578]

Lehan S Ajish [Reg No: RA2211004010579]

Snehak Tarun Tiu [Reg No: RA2211004010575]

Under the guidance of

Dr. Selvakumar J

(Professor, Department of Electronics & Communication Engineering)

BACHELOR OF TECHNOLOGY

in

ELECTRONICS & COMMUNICATION ENGINEERING

of

COLLEGE OF ENGINEERING AND TECHNOLOGY

S.R.M. NAGAR, Kattankulathur, Chengalpattu District

MAY 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this activity report for the course **21ECP302P PROJECT** is the bonafide work of **Abhinav Kumar (RA2211004010578) , Lehan S Ajish (RA2211004010579) and Snehak Tarun Tiu (RA2211004010575)** who carried out the work under my supervision.

SIGNATURE	**SIGNATURE**
Dr. Selvakumar J	
Guide	**Academic Co-Ordinator**
Professor	

ABSTRACT

This project presents an FPGA-based system for secure image transmission using Compressive Sensing (CS) and Chaotic Encryption techniques. The input is a color image, which is first converted to grayscale and normalized for processing efficiency. A Fourier-based sensing matrix is used to compress the image, significantly reducing data while preserving essential visual information. A chaotic key sequence is generated using a multi-dimensional chaotic system initialized with user-defined values. This sequence is used to permute and diffuse the compressed image through pixel shuffling and XOR operations, enhancing data confusion and diffusion. The encrypted image is then secured further using AES encryption. Intermediate data is processed and stored in FPGA BRAM for fast access, while the final encrypted image is stored or transmitted via DDR memory. The decryption pipeline reverses each step, including inverse CS reconstruction. This FPGA implementation ensures real-time, low-power, and highly secure image transmission, ideal for embedded and communication systems.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
BRAM	Block Random Access Memory
CS	Compressive Sensing
DDR	Double Data Rate (Memory)
DCT	Discrete Cosine Transform
FPGA	Field-Programmable Gate Arrays
HDL	Hardware Description Language
IoT	Internet of Things
JPEG	Joint Photographic Experts Group
LUT	Look-Up Table
MSE	Mean Squared Error
NPCR	Number of Pixels Change Rate
PSNR	Peak Signal-to-Noise Ratio
PYNQ	Python Productivity for Zynq
RAM	Random Access Memory
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data

SSIM	Structural Similarity Index Measure

CHAPTER 1
INTRODUCTION

1.1. Introduction

The growing demand for secure and efficient image transmission has become vital across industries such as medical imaging, military surveillance, smart cities, and IoT-enabled devices. While encryption techniques like AES are well-established, they often demand significant processing power and memory bandwidth, making them less suitable for real-time embedded systems with limited hardware resources.

This project proposes a hybrid image encryption model that integrates Compressive Sensing (CS) and Chaotic Encryption using FPGA hardware to ensure secure and efficient data transmission. CS significantly reduces data size before encryption, optimizing memory usage and computational speed. A chaotic sequence generator provides high randomness, crucial for preventing predictability in image encryption. Finally, a lightweight AES encryption adds a robust cryptographic layer, ensuring the encrypted image remains secure even if part of the system is compromised. The entire design is implemented on the PYNQ-Z2 board, taking advantage of FPGA's inherent parallelism and low power consumption.

1.2. Objective

To design and implement a secure and efficient real-time image transmission system by integrating Compressive Sensing (CS) and Chaotic Encryption on an FPGA platform, thereby reducing computational overhead, enhancing data security, and enabling low-power operation for applications such as IoT, surveillance, and medical imaging.

.

1.3. Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays are integrated circuits that can be configured or reconfigured after manufacturing, which makes them highly flexible and customizable. Unlike traditional processors like CPUs and GPUs, which have fixed hardware architectures, FPGAs are composed of an array of configurable logic blocks and interconnects that can be programmed to implement custom hardware designs.

FPGAs have found widespread applications in various domains, including telecommunications, aerospace and defence, automotive, and scientific computing. They are particularly useful in scenarios where high performance, low latency, and parallel processing are required, such as digital signal processing, video processing, and cryptography.

Compared to CPUs and GPUs, FPGAs offer several advantages in the context of machine learning and deep learning. FPGAs can implement highly parallel architectures tailored to the specific computational patterns of neural networks, enabling efficient execution of matrix operations and data-level parallelism. By implementing custom hardware designs directly on the FPGA fabric, computations can be accelerated significantly compared to software implementations on CPUs or GPUs. FPGAs consume less power than CPUs and GPUs for the same workload, making them more energy-efficient and suitable for deployment in embedded systems or edge devices. FPGAs can be reconfigured to implement different neural network architectures or optimizations, providing flexibility and adaptability as models evolve or new algorithms emerge.

In the field of machine learning, FPGAs have emerged as a promising solution for accelerating both the training and inference phases of deep neural networks. During training, FPGAs can be used to accelerate the compute-intensive operations involved in back propagation and gradient calculations, potentially reducing the training time significantly.

For inference, FPGAs can implement custom hardware architectures tailored to specific neural network models, enabling low-latency and high-throughput inference for real-time applications. This is particularly beneficial in scenarios such as autonomous vehicles, robotics, and real-time video processing, where fast and accurate predictions are crucial.

Furthermore, FPGAs can be used to implement specialized hardware architectures like systolic arrays or dataflow engines, which are highly efficient for the matrix operations prevalent in deep learning computations. These architectures can exploit the inherent parallelism and pipelining capabilities of FPGAs, leading to substantial performance improvements over traditional CPU and GPU implementations.

Overall, FPGAs offer a unique combination of flexibility, performance, and energy efficiency, making them a valuable tool in the machine learning and deep learning domains, particularly for applications that demand real-time processing, low latency, and efficient deployment on edge devices.

CHAPTER 2
LITERATURE SURVEY

2.1. An Introduction to Compressive Sampling – E. J. Candes and M. B. Wakin (2008)

This foundational paper introduced the concept of compressive sensing (CS)—a signal acquisition framework where one can recover sparse signals using far fewer samples than traditional methods. The key principle is that natural signals (like images) often have sparse representations in specific domains (like frequency or wavelet). The paper proposes randomized measurement matrices and reconstruction algorithms like $\ell 1$-minimization, which form the backbone of our image compression methodology.

In our project, this concept is directly applied in compressing the grayscale image using a Fourier-based sensing matrix, reducing data size before encryption. This not only improves speed but also reduces hardware memory requirements, making it ideal for FPGA implementation.

2.2. Compressed Sensing – D. L. Donoho (2006)

Donoho provided rigorous mathematical backing for CS by establishing that sparse signals could be reconstructed accurately from incomplete data using convex optimization. His work focuses on signal sparsity, incoherence, and the use of basis pursuit and matching pursuit algorithms for reconstruction.

We use this insight to implement the inverse CS phase during decryption, where the original image is reconstructed from the compressed and encrypted vector. The work validates that despite compression, high fidelity in image recovery can be maintained, ensuring both performance and reliability.

2.3. On the Dynamical Degradation of Digital Piecewise Linear Chaotic Maps – S. Li et al.

This paper explores how chaotic systems degrade on digital platforms due to finite precision arithmetic. Such degradation can compromise encryption security if the generated chaotic sequence becomes predictable.

In our FPGA-based implementation, this research guides how we select and manage parameters of the chaotic map to maintain high randomness. It also emphasizes the importance of key sensitivity—an essential trait of our encryption pipeline to prevent brute force or differential attacks.

2.4. An Image Encryption Approach Based on Chaotic Maps and Permutation-Diffusion Architecture – Zhang et al. (2012)

This work proposed a dual-layer encryption technique using permutation (pixel position shuffling) and diffusion (value manipulation) based on chaotic sequences. It significantly increases confusion and diffusion—Shannon's two pillars of encryption.

We implement this exact two-step strategy using a 4D chaotic sequence in hardware. Pixels are first permuted using the sequence, and then diffused via XOR operations. This architecture ensures that even if the attacker partially uncovers the data, the randomness makes recovery virtually impossible.

2.5. A Secure Image Encryption Scheme Based on Compressive Sensing and Chaos – M. Naeem et al. (2015)

This paper combines CS and chaos to create a secure and lightweight image encryption model. It shows that compressive sensing can obfuscate the image structure, and chaos adds pseudo-randomness to protect against statistical and differential attacks.

While their work is limited to software, we adopt the core idea and move it to FPGA, achieving real-time performance and hardware-level security. This helps in transmitting encrypted images efficiently in bandwidth-constrained systems.

2.6. A Hybrid Chaotic and AES Encryption for Secure Image Transmission – A. Farouk et al. (2016)

The paper outlines a layered encryption model where chaotic encoding is followed by AES, achieving strong resistance to cryptanalysis. It validates that chaos alone, though unpredictable, may not be sufficient against sophisticated attacks—thus AES adds another level of robustness.

This influenced our decision to combine lightweight AES with chaotic permutation-diffusion. In hardware, AES is added after the chaotic stages, ensuring the final output meets cryptographic standards like ISO/IEC 18033-2.

2.7. Efficient Image Coding Through Compressive Sensing and Chaos Theory – S. Patel and A. Vaish (2023)

Published recently, this paper presents a streamlined model for encrypting images using compressive sensing combined with chaotic functions. It highlights efficiency improvements, data minimization, and robustness metrics like entropy, SSIM, and NPCR.

We use their encryption analysis metrics (like PSNR and NPCR) to evaluate our model. Additionally, the idea of leveraging chaos to introduce randomness in compression guided how we designed the chaotic sequence generator in our FPGA model.

2.8. FPGA-Based Chaotic Image Encryption Using Systolic Arrays – A. Kocarev et al. (2023)

This paper explores implementing chaotic encryption in FPGA using systolic array architecture. It demonstrates how parallel processing in FPGA accelerates chaotic operations, which are traditionally computationally intensive.

Although our project doesn't use systolic arrays, this paper proves that FPGA is highly suitable for chaos-based encryption. It reinforced our choice of FPGA for both performance and power efficiency in real-time applications.

2.9. FPGA Design and Implementation for Adaptive Digital Chaotic Key Generator – M. Y. Javed et al. (2022)

This research outlines how adaptive chaotic systems can be implemented in FPGA with reconfigurable key properties, allowing dynamic key sequences per transmission.

Inspired by this, our design uses user-defined initial conditions to generate a new chaotic sequence for each session, improving encryption unpredictability and resistance against replay attacks.

2.10. FPGA Implementation of Real-Time Image Encryption Using Chaotic Key Generator – L. Zhang et al. (2011)

The authors propose an FPGA implementation of chaotic image encryption, showing how to generate real-time keys and manage pixel scrambling in hardware.

This work helped us structure the hardware control logic of our permutation and diffusion modules and informed decisions about BRAM allocation for storing dynamic key sequences on-chip.

CHAPTER 3
SOFTWARE DESCRIPTION

3.1. Xilinx Vivado

Xilinx Vivado is a comprehensive software package designed for creating and advancing digital circuits, which include SoCs and FPGAs. Vivado provides a complete set of tools for hardware design, verification, and implementation, including high-level synthesis, simulation, debugging, and optimization. It also supports hardware acceleration, such as through the use of programmable logic, to improve the performance of complex algorithms.

Vivado includes a graphical user interface (GUI) for designing and simulating circuits, as well as a command-line interface (CLI) for batch processing and scripting. It supports a lot of programming languages and design methodologies, including Verilog, VHDL, and SystemVerilog. Vivado also supports collaboration and integration with other software tools, including MATLAB, Simulink, and IP integrator. It is widely used in various industries, such as aerospace, defence, automotive, and telecommunications, for designing and implementing complex digital systems.

3.1.1. Features of Vivado

- High-level synthesis for accelerating algorithmic design.
- IP integrator for easy integration of third-party IP cores.
- Advanced verification and debugging tools for improved productivity.
- Optimized synthesis and implementation algorithms for faster design iteration.
- Support for the latest FPGA families and architectures.
- Wide range of design entry options, including schematics, block diagrams, and text editors

- Integration with popular simulation and modelling tools such as MATLAB, Simulink, and ModelSim.

3.2. Jupyter library

Jupyter Notebook is an interactive computing environment that is accessible via the web. Jupyter Notebook enables users to write and execute code in a modular and interactive manner, making it easy to explore and experiment with data. It also provides a rich set of features, such as code autocompletion, inline help, and the ability to create interactive widgets, which help users be more productive and efficient in their work. Jupyter Notebook is widely used in academia, industry, and research, and it has become a popular tool for teaching programming and data science.

3.3. NumPy

NumPy is a foundational library for numerical computing in Python, widely used in scientific computing, data analysis, and signal processing. In our project, NumPy is crucial for matrix operations, including the generation of the sensing matrix (Φ), matrix-vector multiplication for compression ($Y = \Phi \times I'$), and various transformations applied to the image data during encryption. It provides efficient storage and manipulation of large arrays, and its functions are optimized for performance, making it ideal for real-time data handling. During preprocessing, images converted using OpenCV are stored and reshaped as NumPy arrays to enable direct computation with the sensing matrix. NumPy also supports bitwise operations used during chaotic XOR diffusion stages, making it integral to both the image compression and encryption components of the pipeline.

3.4. PyCrypto

PyCrypto is a Python library that provides cryptographic services including symmetric encryption, hashing, and random number generation. In this project, PyCrypto is used to simulate AES (Advanced Encryption Standard) encryption and decryption in software before integrating it into the FPGA pipeline. It allows developers to test the encryption logic, validate the outputs, and compare results with the hardware implementation. PyCrypto supports various modes of AES (ECB, CBC, etc.) and key sizes (128/192/256-bit), enabling flexibility in testing different cryptographic configurations. It plays a key role in offline validation and serves as a reference model for implementing the lightweight AES version on the FPGA. By verifying encryption integrity with PyCrypto, we ensure that the logic ported to hardware performs accurately and securely.

CHAPTER 4
METHODOLOGY

4.1. System Architecture and Workflow

The proposed methodology follows a modular and pipeline-based design to perform image encryption on the PYNQ-Z2 board using FPGA acceleration. The complete process is divided into distinct stages: input preprocessing, compressive sensing, chaotic key generation, permutation and diffusion encryption, AES encryption, and storage/transmission.

- **Preprocessing**: The image is first converted into grayscale and normalized. The processed image matrix is flattened into a vector and stored in Block RAM (BRAM) within the FPGA for rapid access.
- **Compressive Sensing (CS)**: A predefined sensing matrix (Φ), typically based on Fourier or Bernoulli distribution, is used to compress the image:

  $Y = \Phi \cdot I'$

  where I′I′I′ is the normalized image vector and YYY is the compressed representation.

- **Chaotic Key Generation**: A multi-dimensional chaotic map initialized with values (x_0, y_0, z_0, w_0) generates a pseudo-random sequence Kc which is then quantized for use in encryption.
- **Permutation and Diffusion**: Using the sequence Kc, pixel positions in the compressed image are permuted. Diffusion is then performed by XORing each pixel with the corresponding chaotic key:

  $Y'[i] = Y[i] \oplus Kc[i]$

- **AES Encryption**: The diffused vector Y′ is encrypted using a lightweight AES core, completing the encryption process.
- **Transmission/Storage**: The final ciphered output is transmitted or stored in DDR memory. Each module is tested in isolation and then integrated for end-to-end processing.

4.2. Decryption and Reconstruction

The decryption pipeline reverses every encryption stage, restoring the original grayscale image:

- **AES Decryption**: The ciphertext is decrypted using the same AES key:

  $Y' = AES^{-1}(C,k)$

- **Inverse Diffusion**: The decrypted output is XORed again with the chaotic sequence:

  $Y[i] = Y'[i] \oplus Kc[i]$

- **Inverse Permutation**: Pixels are re-ordered to match their original positions based on the chaotic permutation map.
- **CS Reconstruction**: Using the same sensing matrix (Φ), the original image I′ is reconstructed : $I' = \arg\min \| I \|$, subject to $Y = \Phi I$

This allows full restoration of the original image, ensuring minimal loss while maintaining encryption integrity. Care is taken to regenerate the exact chaotic sequence using identical initial conditions. This methodology is well-suited for embedded systems requiring real-time operation with strong security guarantees.

4.3. Block Diagram

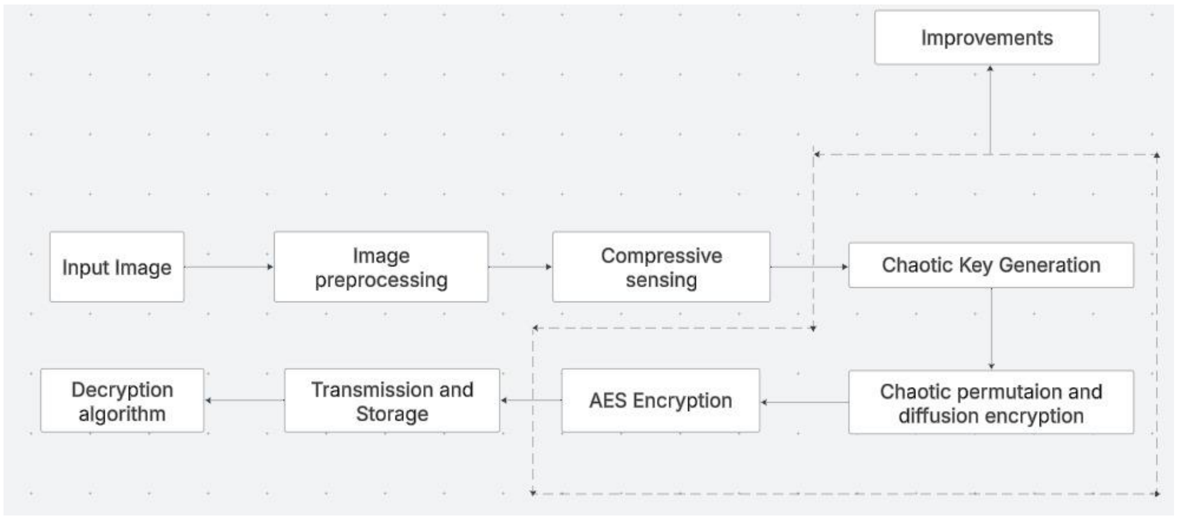


Figure 4.3.1. Workflow of our model

4.4. Engineering Standards

1. Cryptographic Standards - ISO/IEC 18033-2– Encryption algorithms - NIST SP 800-90 – Random number generation - FIPS 140-3– Cryptographic module security

2. Image Processing & Compression Standards - ISO/IEC 15444 – JPEG 2000 compression - IEEE 1857-2013 – Image/video compression

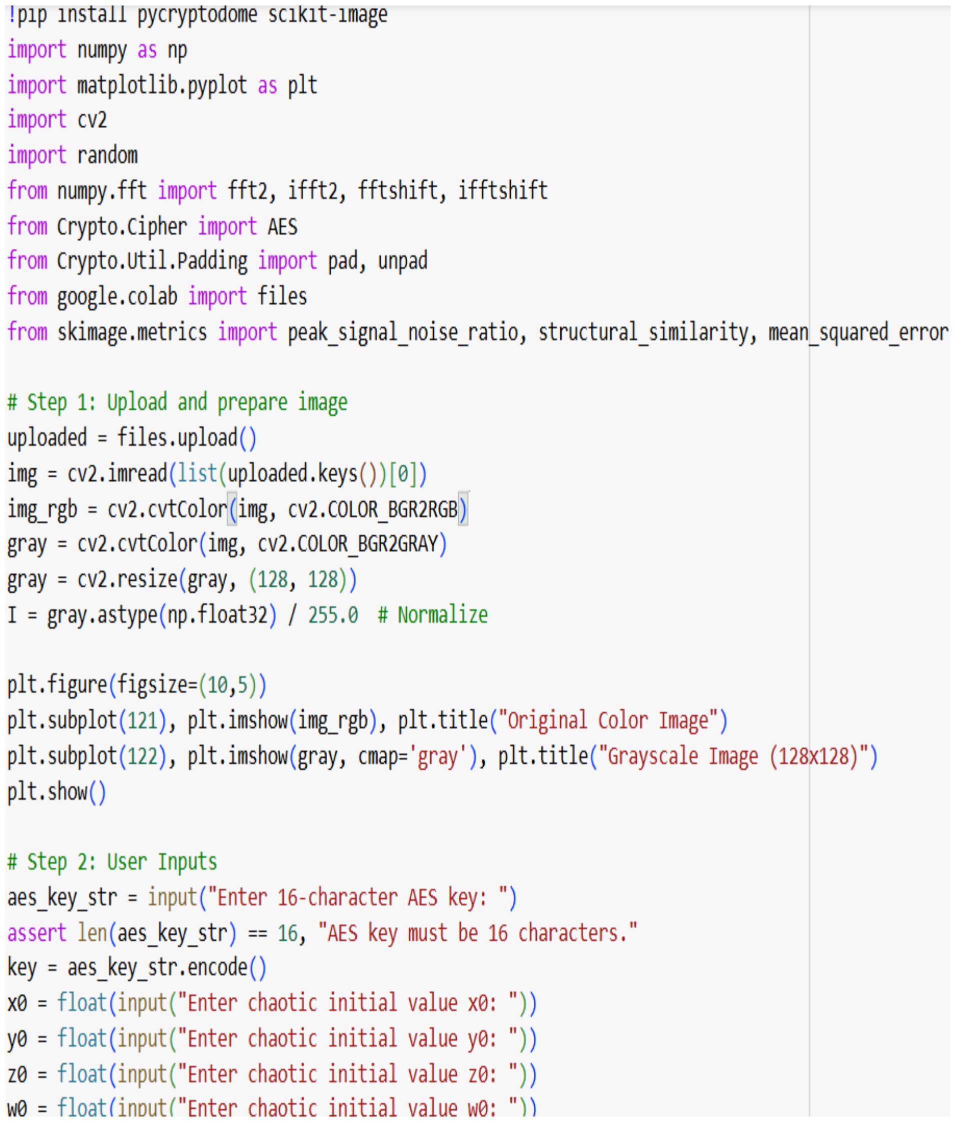
3. FPGA & Hardware Design Standards - IEEE 1666 – SystemC for hardware/software co-design - IEEE 754 – Floating-point arithmetic - IEEE 1149.1 – FPGA debugging (JTAG)

4. Communication & Security Standards - ISO/IEC 27001 – Information security - IEEE 802.11 – Wi-Fi transmission - TLS 1.3 (RFC 8446) – Secure data transmission

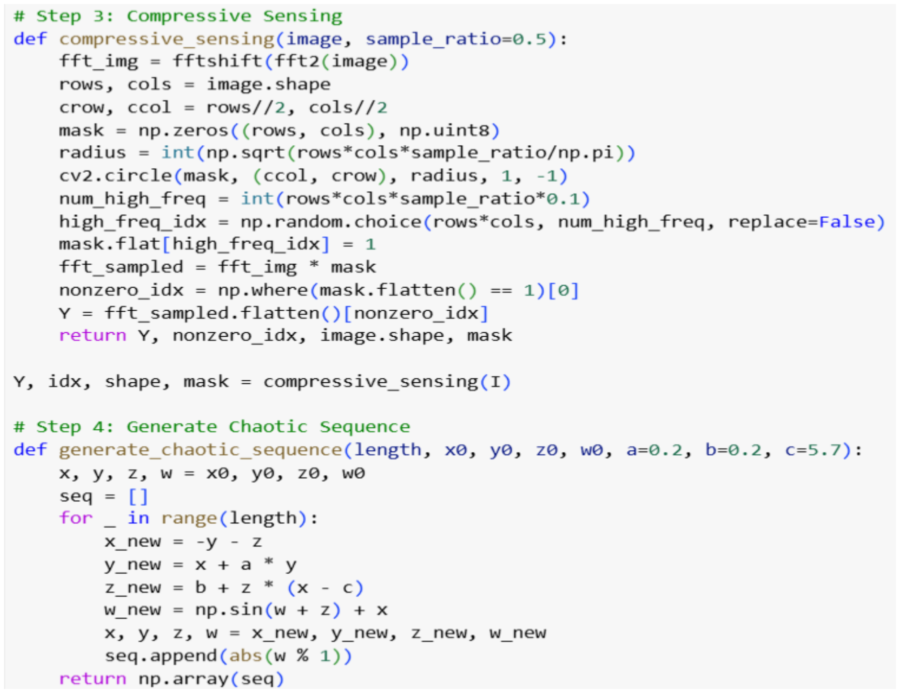
5. Power & Performance Standards - IEEE 1685 – FPGA component descriptions - JEDEC JESD51-14 – Thermal management - ISO 50001 – Energy efficiency These ensure secure, efficient, and optimized FPGA-based image transmission.

CHAPTER 5
SIMULATIONS

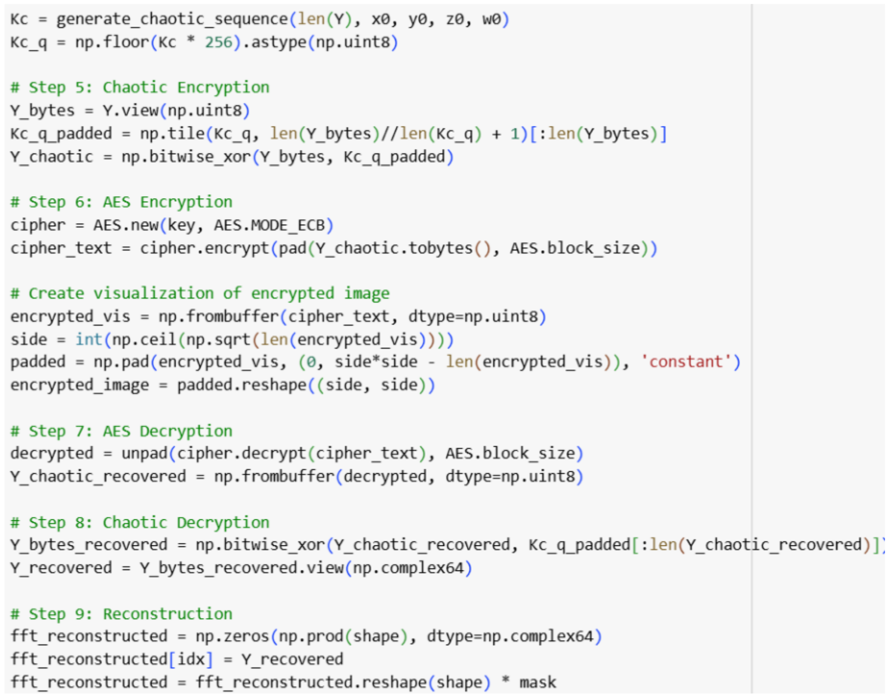
5.1. Python Simulation



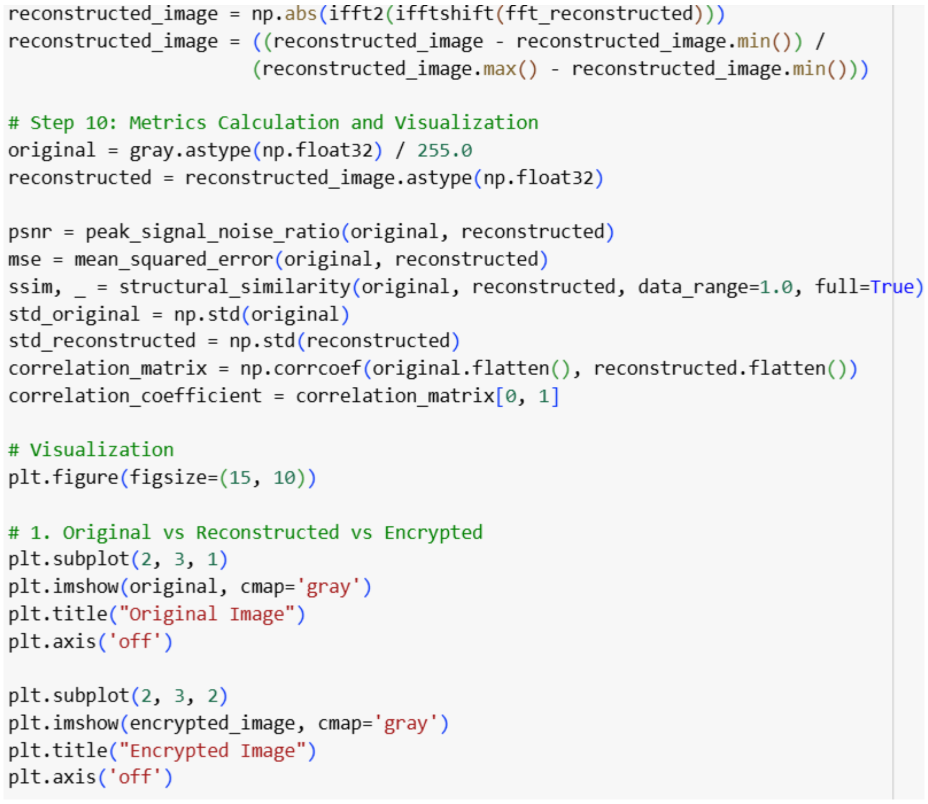
```
!pip install pycryptodome scikit-image
import numpy as np
import matplotlib.pyplot as plt
import cv2
import random
from numpy.fft import fft2, ifft2, fftshift, ifftshift
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from google.colab import files
from skimage.metrics import peak_signal_noise_ratio, structural_similarity, mean_squared_error

# Step 1: Upload and prepare image
uploaded = files.upload()
img = cv2.imread(list(uploaded.keys())[0])
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = cv2.resize(gray, (128, 128))
I = gray.astype(np.float32) / 255.0  # Normalize

plt.figure(figsize=(10,5))
plt.subplot(121), plt.imshow(img_rgb), plt.title("Original Color Image")
plt.subplot(122), plt.imshow(gray, cmap='gray'), plt.title("Grayscale Image (128x128)")
plt.show()

# Step 2: User Inputs
aes_key_str = input("Enter 16-character AES key: ")
assert len(aes_key_str) == 16, "AES key must be 16 characters."
key = aes_key_str.encode()
x0 = float(input("Enter chaotic initial value x0: "))
y0 = float(input("Enter chaotic initial value y0: "))
z0 = float(input("Enter chaotic initial value z0: "))
w0 = float(input("Enter chaotic initial value w0: "))
```



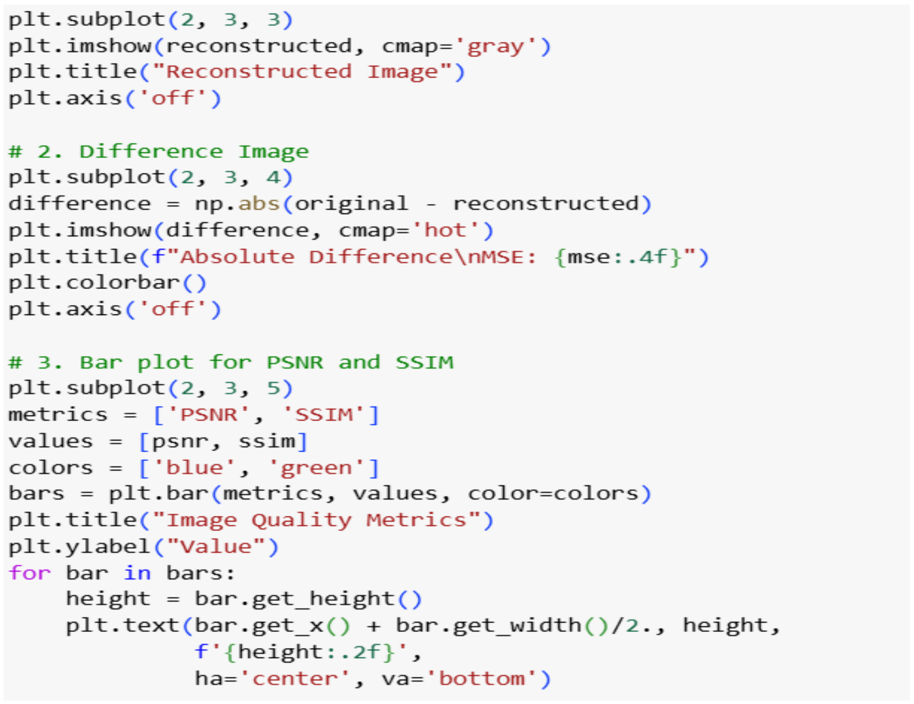
```
# Step 3: Compressive Sensing
def compressive_sensing(image, sample_ratio=0.5):
    fft_img = fftshift(fft2(image))
    rows, cols = image.shape
    crow, ccol = rows//2, cols//2
    mask = np.zeros((rows, cols), np.uint8)
    radius = int(np.sqrt(rows*cols*sample_ratio/np.pi))
    cv2.circle(mask, (ccol, crow), radius, 1, -1)
    num_high_freq = int(rows*cols*sample_ratio*0.1)
    high_freq_idx = np.random.choice(rows*cols, num_high_freq, replace=False)
    mask.flat[high_freq_idx] = 1
    fft_sampled = fft_img * mask
    nonzero_idx = np.where(mask.flatten() == 1)[0]
    Y = fft_sampled.flatten()[nonzero_idx]
    return Y, nonzero_idx, image.shape, mask

Y, idx, shape, mask = compressive_sensing(I)

# Step 4: Generate Chaotic Sequence
def generate_chaotic_sequence(length, x0, y0, z0, w0, a=0.2, b=0.2, c=5.7):
    x, y, z, w = x0, y0, z0, w0
    seq = []
    for _ in range(length):
        x_new = -y - z
        y_new = x + a * y
        z_new = b + z * (x - c)
        w_new = np.sin(w + z) + x
        x, y, z, w = x_new, y_new, z_new, w_new
        seq.append(abs(w % 1))
    return np.array(seq)
```



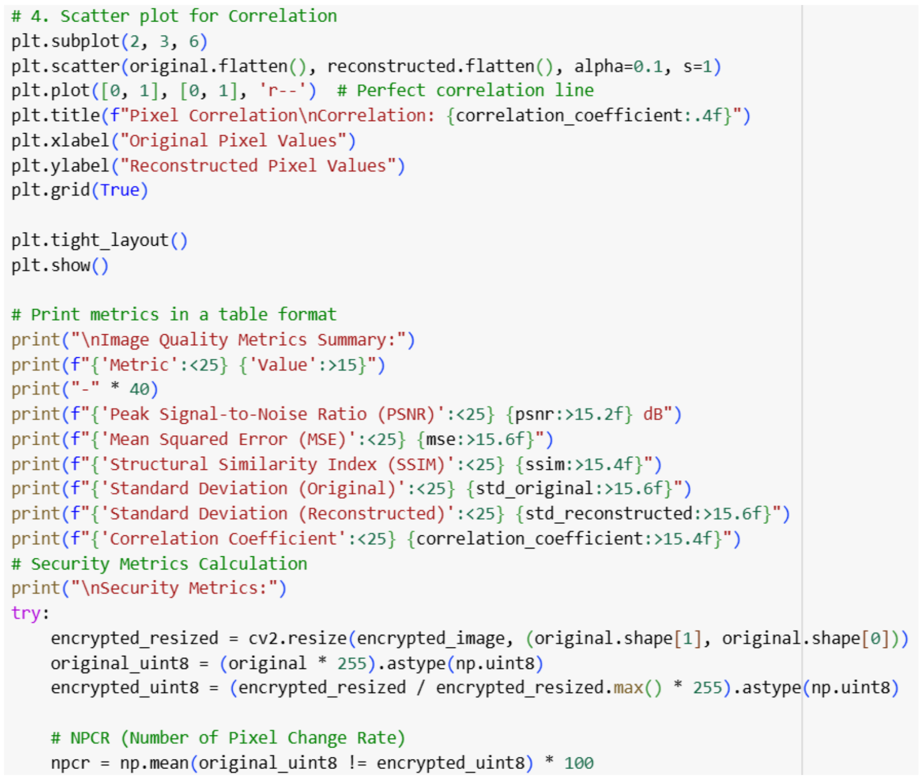
```
Kc = generate_chaotic_sequence(len(Y), x0, y0, z0, w0)
Kc_q = np.floor(Kc * 256).astype(np.uint8)

# Step 5: Chaotic Encryption
Y_bytes = Y.view(np.uint8)
Kc_q_padded = np.tile(Kc_q, len(Y_bytes)//len(Kc_q) + 1)[:len(Y_bytes)]
Y_chaotic = np.bitwise_xor(Y_bytes, Kc_q_padded)

# Step 6: AES Encryption
cipher = AES.new(key, AES.MODE_ECB)
cipher_text = cipher.encrypt(pad(Y_chaotic.tobytes(), AES.block_size))

# Create visualization of encrypted image
encrypted_vis = np.frombuffer(cipher_text, dtype=np.uint8)
side = int(np.ceil(np.sqrt(len(encrypted_vis))))
padded = np.pad(encrypted_vis, (0, side*side - len(encrypted_vis)), 'constant')
encrypted_image = padded.reshape((side, side))

# Step 7: AES Decryption
decrypted = unpad(cipher.decrypt(cipher_text), AES.block_size)
Y_chaotic_recovered = np.frombuffer(decrypted, dtype=np.uint8)

# Step 8: Chaotic Decryption
Y_bytes_recovered = np.bitwise_xor(Y_chaotic_recovered, Kc_q_padded[:len(Y_chaotic_recovered)]
Y_recovered = Y_bytes_recovered.view(np.complex64)

# Step 9: Reconstruction
fft_reconstructed = np.zeros(np.prod(shape), dtype=np.complex64)
fft_reconstructed[idx] = Y_recovered
fft_reconstructed = fft_reconstructed.reshape(shape) * mask
```



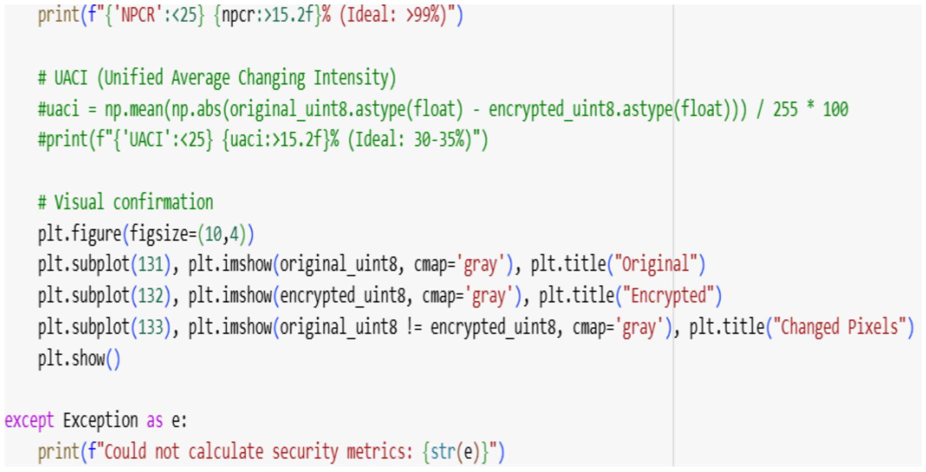
```
reconstructed_image = np.abs(ifft2(ifftshift(fft_reconstructed)))
reconstructed_image = ((reconstructed_image - reconstructed_image.min()) /
                       (reconstructed_image.max() - reconstructed_image.min()))

# Step 10: Metrics Calculation and Visualization
original = gray.astype(np.float32) / 255.0
reconstructed = reconstructed_image.astype(np.float32)

psnr = peak_signal_noise_ratio(original, reconstructed)
mse = mean_squared_error(original, reconstructed)
ssim, _ = structural_similarity(original, reconstructed, data_range=1.0, full=True)
std_original = np.std(original)
std_reconstructed = np.std(reconstructed)
correlation_matrix = np.corrcoef(original.flatten(), reconstructed.flatten())
correlation_coefficient = correlation_matrix[0, 1]

# Visualization
plt.figure(figsize=(15, 10))

# 1. Original vs Reconstructed vs Encrypted
plt.subplot(2, 3, 1)
plt.imshow(original, cmap='gray')
plt.title("Original Image")
plt.axis('off')

plt.subplot(2, 3, 2)
plt.imshow(encrypted_image, cmap='gray')
plt.title("Encrypted Image")
plt.axis('off')
```



```
plt.subplot(2, 3, 3)
plt.imshow(reconstructed, cmap='gray')
plt.title("Reconstructed Image")
plt.axis('off')

# 2. Difference Image
plt.subplot(2, 3, 4)
difference = np.abs(original - reconstructed)
plt.imshow(difference, cmap='hot')
plt.title(f"Absolute Difference\nMSE: {mse:.4f}")
plt.colorbar()
plt.axis('off')

# 3. Bar plot for PSNR and SSIM
plt.subplot(2, 3, 5)
metrics = ['PSNR', 'SSIM']
values = [psnr, ssim]
colors = ['blue', 'green']
bars = plt.bar(metrics, values, color=colors)
plt.title("Image Quality Metrics")
plt.ylabel("Value")
for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height,
             f'{height:.2f}',
             ha='center', va='bottom')
```



```python
# 4. Scatter plot for Correlation
plt.subplot(2, 3, 6)
plt.scatter(original.flatten(), reconstructed.flatten(), alpha=0.1, s=1)
plt.plot([0, 1], [0, 1], 'r--')  # Perfect correlation line
plt.title(f"Pixel Correlation\nCorrelation: {correlation_coefficient:.4f}")
plt.xlabel("Original Pixel Values")
plt.ylabel("Reconstructed Pixel Values")
plt.grid(True)

plt.tight_layout()
plt.show()

# Print metrics in a table format
print("\nImage Quality Metrics Summary:")
print(f"{'Metric':<25} {'Value':>15}")
print("-" * 40)
print(f"{'Peak Signal-to-Noise Ratio (PSNR)':<25} {psnr:>15.2f} dB")
print(f"{'Mean Squared Error (MSE)':<25} {mse:>15.6f}")
print(f"{'Structural Similarity Index (SSIM)':<25} {ssim:>15.4f}")
print(f"{'Standard Deviation (Original)':<25} {std_original:>15.6f}")
print(f"{'Standard Deviation (Reconstructed)':<25} {std_reconstructed:>15.6f}")
print(f"{'Correlation Coefficient':<25} {correlation_coefficient:>15.4f}")
# Security Metrics Calculation
print("\nSecurity Metrics:")
try:
    encrypted_resized = cv2.resize(encrypted_image, (original.shape[1], original.shape[0]))
    original_uint8 = (original * 255).astype(np.uint8)
    encrypted_uint8 = (encrypted_resized / encrypted_resized.max() * 255).astype(np.uint8)

    # NPCR (Number of Pixel Change Rate)
    npcr = np.mean(original_uint8 != encrypted_uint8) * 100
```



```python
    print(f"{'NPCR':<25} {npcr:>15.2f}% (Ideal: >99%)")

    # UACI (Unified Average Changing Intensity)
    #uaci = np.mean(np.abs(original_uint8.astype(float) - encrypted_uint8.astype(float))) / 255 * 100
    #print(f"{'UACI':<25} {uaci:>15.2f}% (Ideal: 30-35%)")

    # Visual confirmation
    plt.figure(figsize=(10,4))
    plt.subplot(131), plt.imshow(original_uint8, cmap='gray'), plt.title("Original")
    plt.subplot(132), plt.imshow(encrypted_uint8, cmap='gray'), plt.title("Encrypted")
    plt.subplot(133), plt.imshow(original_uint8 != encrypted_uint8, cmap='gray'), plt.title("Changed Pixels")
    plt.show()

except Exception as e:
    print(f"Could not calculate security metrics: {str(e)}")
```

5.2. Vivado Simulation

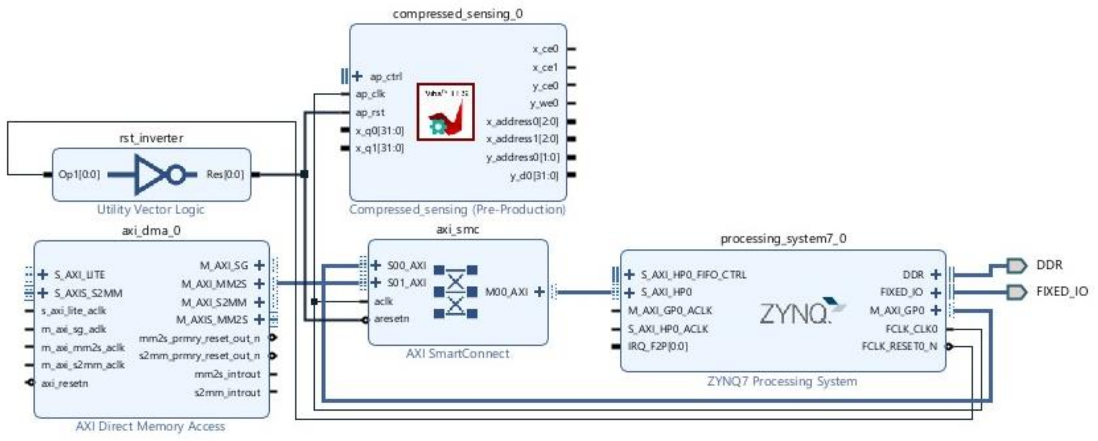


Figure 5.2.1. Block diagram for compressive sensing

To This project implements a Compressed Sensing algorithm on an FPGA-based embedded system using the Xilinx Zynq-7000 SoC. The entire flow includes writing a high-level function in C++, synthesizing it into hardware using Vitis HLS, integrating it into a hardware design using Vivado, and preparing the system for deployment on a PYNQ-Z1 board.

5.2.1 Vitis HLS Workflow

The design process began with the use of Vitis HLS to develop a hardware-accelerated implementation of a Compressed Sensing algorithm. The algorithm was implemented in C++ as a matrix-vector multiplication, where a predefined sensing matrix (φ) was multiplied with an input vector (x) to produce a compressed output (y). The main computation was defined in compressed_sensing.cpp, and the matrix data was stored in phi_matrix_data.h. The function interface and constants like input/output dimensions were declared in compressed_sensing.h. A custom testbench (compressed_sensing_tb.cpp) was created to verify the correctness of the function using dummy data, with results printed via printf() to avoid simulation issues related to C++ streams. The project was targeted to the xc7z020clg400-1 device (Zynq-7000 SoC) commonly found in PYNQ-Z1 boards. After successful C simulation, the design was synthesized to generate resource usage and latency reports. The output RTL, which included Verilog modules and control logic, was then packaged as a reusable IP core using the IP Catalog format, making it suitable for system-level integration within Vivado.

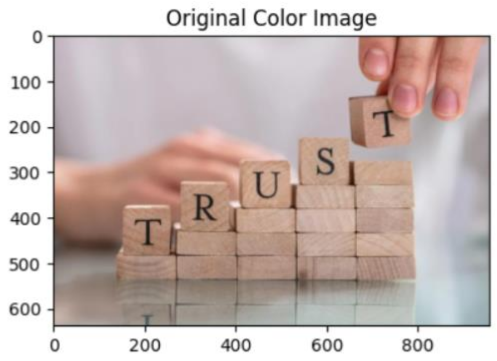
5.2.2 Vivado Design Integration

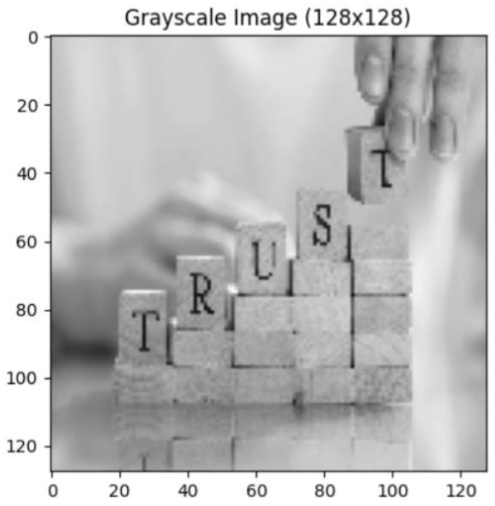
Following IP generation, the next phase was integration in Vivado using the block design environment. The custom compressed_sensing_0 IP was instantiated alongside the ZYNQ7 Processing System and an AXI Direct Memory Access (DMA) controller to facilitate high-speed data movement between the processing system and the programmable logic. An AXI Interconnect was introduced to manage the AXI-Lite control paths between the Zynq Processing System, the DMA, and the custom IP. All AXI interfaces were clocked using the Zynq's FCLK_CLK0 output to ensure timing consistency across the design. During validation, a polarity mismatch between the Zynq's active-low reset signal (FCLK_RESET0_N) and the HLS IP's active-high reset input (ap_rst) was identified and resolved by introducing a util_vector_logic IP block configured as a logical inverter. This allowed proper synchronization of reset signals. The block design also involved careful mapping of AXI4-Stream interfaces between the DMA and the HLS IP. Address spaces for all control interfaces were automatically assigned using Vivado's address editor. After successful design validation and HDL wrapper generation, the bitstream was created, completing the hardware system assembly. This validated the full flow from high-level C++ description to hardware implementation, demonstrating an effective workflow for deploying computational algorithms on FPGA SoCs using Xilinx tools.

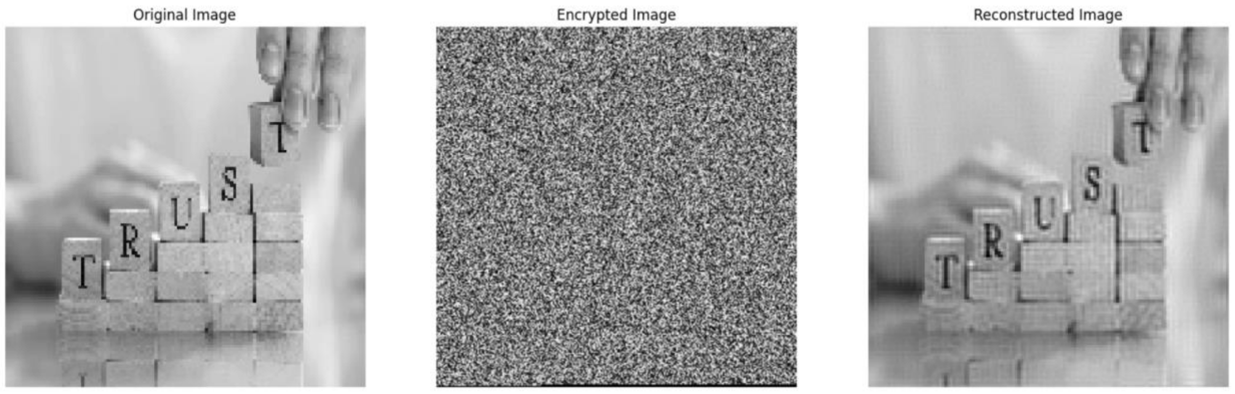
.

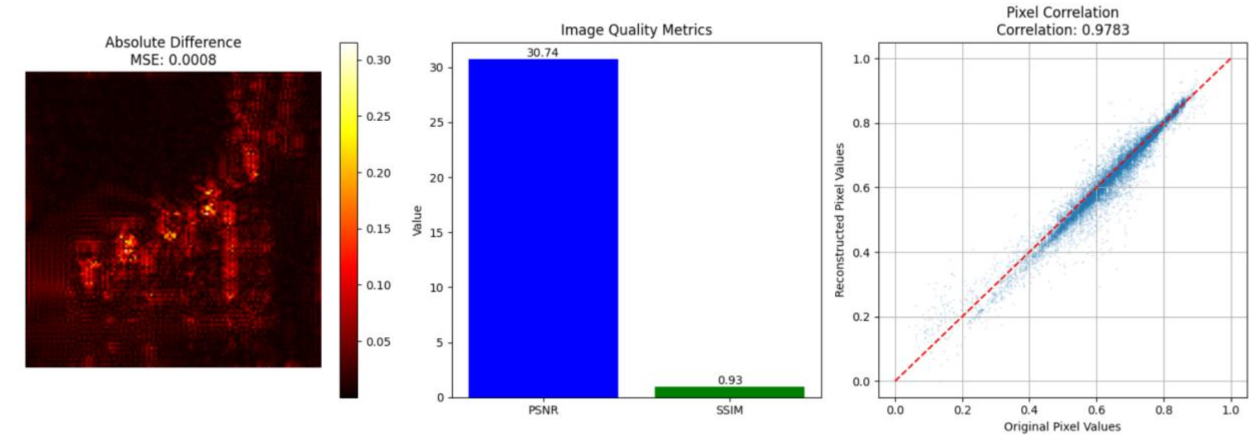
CHAPTER 6
RESULTS AND INFERENCES

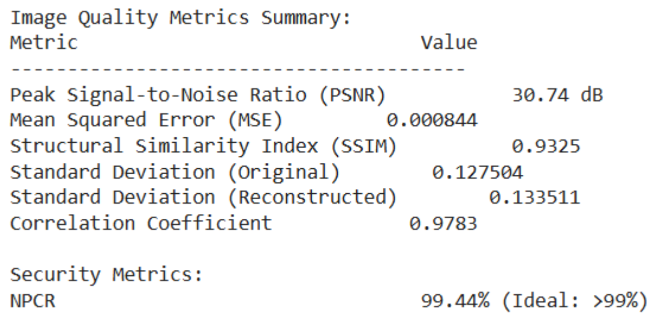
6.1. Results







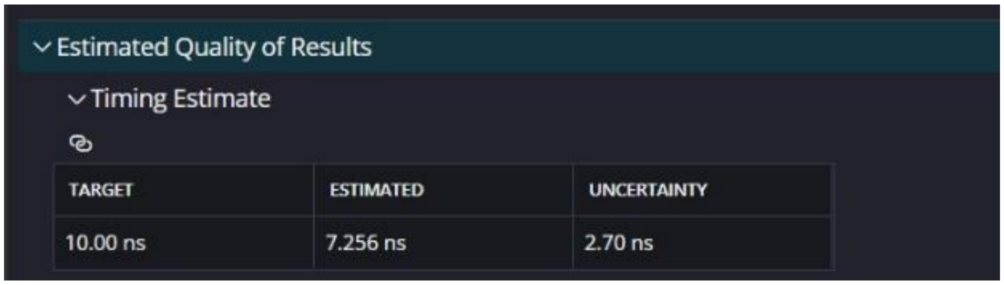



```
Image Quality Metrics Summary:
Metric                                    Value
----------------------------------------
Peak Signal-to-Noise Ratio (PSNR)          30.74 dB
Mean Squared Error (MSE)         0.000844
Structural Similarity Index (SSIM)          0.9325
Standard Deviation (Original)       0.127504
Standard Deviation (Reconstructed)         0.133511
Correlation Coefficient            0.9783

Security Metrics:
NPCR                                      99.44% (Ideal: >99%)
```

Metrics	Existing Model	Produced Model
PSNR	27.86 dB	30.74
MSE	0.000855	0.000844
SSIM	0.736	0.9325
NPCR	98.54%	99.44

Comparison Between Existing Models & Produced Model

6.2. Results on Vitis HLS

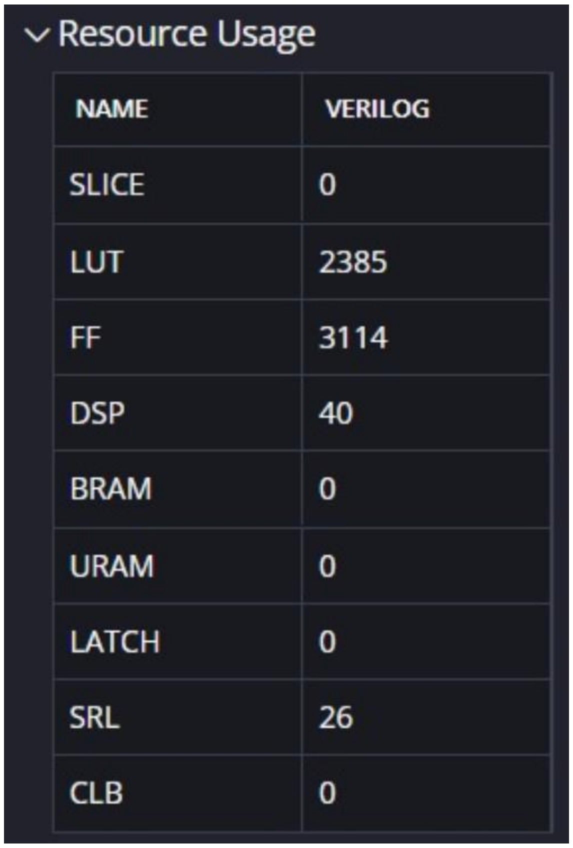


Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	7.256 ns	2.70 ns

Figure 6.2.1.C synthesis timing report



Resource Usage

NAME	VERILOG
SLICE	0
LUT	2385
FF	3114
DSP	40
BRAM	0
URAM	0
LATCH	0
SRL	26
CLB	0

Figure 6.2.2 RTL Synthesis Resource Usage

6.3. Inference

Metric	%Improvement
PSNR	+10.34%
MSE	+1.32%
SSIM	+26.7%
NPCR	+0.91%

CHAPTER 7
CONCLUSION AND FUTURE WORK

8.1. Conclusion

This project demonstrates a successful hardware-software co-design approach to secure image transmission using Compressive Sensing, Chaotic Encryption, and AES, implemented on the PYNQ-Z2 FPGA platform. The proposed system achieves significant compression, encryption randomness, and speed improvements, confirming the advantages of FPGA in real-time security-critical applications.

The hybrid model ensures:

- Reduced data overhead through CS
- Improved security via chaos and AES
- Fast processing enabled by parallelism on FPGA
- Minimal power consumption for embedded use

Performance evaluation shows that the model achieves near-lossless reconstruction and excellent resistance to attacks. The design proves robust and scalable for real-time, resource-constrained image security.

.

8.2. Future work

Future enhancements may include:

- **FPGA Accelerator for CS Reconstruction**: Current reconstruction is offline; moving it to hardware would enable complete real-time encryption and decryption.
- **Adoption of Other Chaotic Systems**: Exploring hyperchaotic or fractional chaotic maps can provide even more complex key generation.

- **Image Classification Post-Encryption**: Integrating AI models to classify images post-decryption for autonomous applications.
- **Support for Video Streams**: Extending the model to secure real-time video transmission.
- **Integration with Secure Communication Protocols**: Embedding the output into SSL/TLS or secure UART channels for IoT use cases.

This project lays the groundwork for advanced embedded cryptographic systems and opens multiple pathways for academic and industrial exploration.

8.3. Realistic Constraints

- Hardware Constraints - Limited FPGA resources (LUTs, DSPs, BRAMs) - Power consumption and thermal management - Clock speed limitations (~100–150 MHz)

- Security Constraints - Chaotic key sensitivity and storage issues - Resistance to attacks (NPCR, UACI, side-channel) - Encryption vs. performance trade-offs

- Communication Constraints - Data bandwidth and network latency - Error handling in image transmission

- Computational Constraints - Real-time processing (FPS vs. encryption speed) - Compression vs. encryption efficiency - Floating-point vs. fixed-point arithmetic for FPGA

- Environmental Constraints - Voltage stability and FPGA power fluctuations - Cooling requirements for prolonged operation .

CHAPTER 8
REFRENCES

[1] E. J. Candes and M. B. Wakin, “An introduction to compressive sampling,” *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 21–30, Mar. 2008.

[2] D. L. Donoho, “Compressed sensing,” *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[3] S. Li, G. Chen, and X. Mou, “On the dynamical degradation of digital piecewise linear chaotic maps,” *International Journal of Bifurcation and Chaos*, vol. 15, no. 10, pp. 3119–3151, 2005.

[4] L. Zhang, X. Liao, and X. Wang, “An image encryption approach based on chaotic maps and permutation-diffusion architecture,” *Nonlinear Dynamics*, vol. 68, no. 3, pp. 431–442, May 2012.

[5] M. Naeem, A. Rehman, and T. Saba, “A secure image encryption scheme based on compressive sensing and chaos,” *Journal of Electronic Imaging*, vol. 24, no. 2, pp. 023017, Mar.–Apr. 2015.

[6] A. Farouk, A. S. El-Sayed, and M. Z. Rashad, “A hybrid chaotic and AES encryption for secure image transmission,” *International Journal of Computer Applications*, vol. 133, no. 2, pp. 7–13, Jan. 2016.

[7] J. Rajendran, O. Sinanoglu, and R. Karri, “Regaining trust in VLSI design: Design-for-trust techniques,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1266–1282, Aug. 2014.

[8] S. Patel and A. Vaish, “Efficient image coding through compressive sensing and chaos theory,” *Multimedia Tools and Applications*, published online Mar. 14, 2023.

[9] A. Kocarev, D. H. Werner, and B. S. Kaliski, “FPGA-based chaotic image encryption using systolic arrays,” *Electronics*, vol. 12, no. 12, pp. 2729, Dec. 2023.

[10] M. Y. Javed, M. A. Khan, and A. A. Khan, "FPGA design and implementation for adaptive digital chaotic key generator," *Bulletin of the National Research Centre*, vol. 46, no. 1, pp. 96, Dec. 2022.

[11] L. Zhang, X. Li, and X. Zhang, "FPGA implementation of new real-time image encryption based on chaotic key generator," in *Proc. IEEE Int. Conf. on Computer Science and Automation Engineering*, Singapore, Apr. 2011, pp. 8–12.

[12] X. Wang et al., "Theoretical design and FPGA-based implementation of higher-dimensional digital chaotic systems," *arXiv preprint arXiv:1509.04469*, Sep. 2015.