Enhancing Robustness of Digital Systems Against Hardware Trojan Attacks Using Stochastic Computing

Andrew Han[2] (ajh307) Camden Larson[1] (csl117) William Froass[2] (wpf10)

Technical Advisor: Dr. M. Hassan Najafi, Assistant Professor, Department of Electrical, Computer, and Systems Engineering, Case School of Engineering

Instructor: Dr. Vira Chankong

Submitted for ECSE 398 in partial fulfillment for the degree of Bachelor of Science of

Computer Engineering[1] |

Electrical Engineering[2]

Case Western Reserve University

Fall 2025

Academic Integrity Policy

All students in this course are expected to adhere to University standards of academic integrity. Cheating, plagiarism, misrepresentation, and other forms of academic dishonesty will not be tolerated. This includes, but is not limited to, consulting with another person during an exam, turning in written work that was prepared by someone other than you, making minor modifications to the work of someone else and turning it in as your own, or engaging in misrepresentation in seeking a postponement or extension. Ignorance will not be accepted as an excuse. If you are not sure whether something you plan to submit would be considered either cheating or plagiarism, it is your responsibility to ask for clarification. For complete information, please go to https://students.case.edu/community/conduct/aiboard/policy.html.

The individual(s) submitting this document affirm compliance with the above statement:

[This page intentionally left blank

Executive Summary

In this project, we designed a stochastic implementation of an image processing algorithm called the Roberts Edge Detector. Our goal was to create a more resilient implementation than traditional binary ones by applying stochastic computing in order to make the Edge Detector maintain accuracy even when facing Hardware Trojan Attacks. We successfully created a stochastic Roberts Edge Detector that maintained a higher accuracy and peak signal-to-noise ratio than a binary Roberts Edge Detector. Although we did not have the time to verify this fact, we believe that stochastic implementation of systems are also less complex and use less energy.

Table of Contents

[This page intentionally left blank]

Problem Statement

Modern digital systems are vulnerable to Hardware Trojan Attacks (HTA), which are manipulations of the inputs to these digital systems made to cause data leaks or systematic failure. This vulnerability is largely due to the binary architecture of digital systems, which allows for certain combinations of inputs to cause adverse and potentially catastrophic outputs. As circuits become more complex, they grow even more vulnerable to HTAs because they are difficult to manually inspect, making it harder to ensure that no harmful modifications are present.

We will design an implementation of digital systems using stochastic computing in order to make them more resilient against HTAs. The digital system we will be focusing on in this project is an image processing algorithm called the Roberts Edge Detector. We will implement the binary version of the edge detector and design a stochastic version of the edge detector at the gate level. After designing both implementations, we will compare how they perform with and without HTAs both visually by displaying the images and quantitatively by comparing their quality and accuracy.

Our stochastic design will maintain a higher accuracy against HTAs than traditional binary designs do. By designing digital systems with stochastic computing, these systems will have improved performance accuracy when facing successful HTAs without needing to detect them. In addition, digital systems using stochastic computing will have lower gate complexity and use less energy.

Current defense methods against HTAs rely on detecting and counteracting the manipulations made to the systems. Methods such as redundancy-based designs, physically unclonable functions, and software obfuscation offer significant protections against hardware trojan attacks, but introduce significant trade-offs in terms of area, power consumption, and cost. These attacks also lack the ability to maintain accuracy in the presence of successful attacks.

Purpose, Goals, & Objectives

This project intends to design a stochastic implementation of the Roberts Edge Detector in order to increase its resilience against HTAs. A stochastic implementation can be designed for important systems beyond the Roberts Edge Detector such as in finance and energy. By enabling these systems to maintain high performance even when facing successful HTAs, stochastic computing will make these systems much more secure and processes in the world will run smoothly with less malicious, intentional disruptions. In addition, these systems will use less hardware and energy and not have to be designed around detecting and fighting HTAs.

To show that our stochastic design poses a solution, our stochastic Roberts Edge Detector must correctly produce edge detecting images, maintain performance against HTAs, and rely on low hardware complexity.

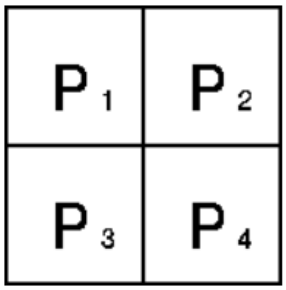
Background & Context

Stochastic computing performs operations on stochastic bitstreams, where a number is encoded as a the probability of 1s in the bitstream (e.g., 100101 has 3 ones out of 6, representing 0.5). The probabilistic nature of stochastic computing makes it resilient to HTAs. Manipulations to individual bits in a stochastic

bitstream have a small impact on the final computed value as compared to binary bitstreams. HTAs often rely on altering or injecting bits into data streams. Stochastic computing also provides a significant reduction in hardware complexity. Stochastic addition can be implemented using a single MUX, stochastic subtraction only requires an XOR gate, and stochastic multiplication can be performed using only an AND gate. Stochastic bitstreams are generated using Sobol sequences, and the streams are intentionally correlated to improve computational stability.

Hardware trojan attacks are malicious, unauthorized, or stealthy modifications inserted into integrated circuits that alter functionality, reduce reliability, or leak sensitive information. They typically remain dormant during normal operation and are active under certain conditions or circumstances. We focused on a gate level hardware trojan attack in which an extra MUX is inserted on the output of an operation. This trojan simply passes the normal output through most of the time, but can be activated based on a variety of events. The purpose of this malicious additional MUX is to pass in a Trojan value instead of the expected bit at some arbitrary frequency effectively altering the output of the operation while remaining unnoticed in large scale systems.

The Roberts Edge Detector is an algorithm that displays the edges of an images in a new image. Edges are a measure of the intensity gradient between neighboring pixels in an image. Higher gradients are shown as lighter in the processed image while lower gradients are shown as darker. The Roberts Edge Detector is used in fields such as healthcare and power systems.



In the Roberts Edge Detector, edge gradient is calculated in 2x2 kernels of pixels as shown above. Components of gradient are calculated as $G_x = |P_1 - P_4|$ and $G_y = |P_2 - P_3|$. The overall gradient is calculated as $|G| = \sqrt{Gx^2 + Gy^2}$ but can be approximated as $|G| = |Gx| + |Gy|$. In our implementations, we will be using the approximated method.

Success Criteria

To be considered a success, the project must demonstrate that our stochastic computing design provides improved robustness against HTAs compared to traditional binary designs. The following quantitative criteria will be used to assess success:

1. Accuracy Under Trojan Attack
    - The stochastic design must achieve a higher accuracy than the binary design when both are subjected to the same hardware trojan attacks.
    - Success Threshold:

$$\text{Accuracy}_{\text{stochastic}} > \text{Accuracy}_{\text{binary}}$$

2. Edge Detection Performance Under Trojan Attack

- The stochastic Roberts edge detection design must outperform the binary edge detection design when both are subjected to the same hardware trojan attacks.
- Peak Signal-to-Noise Ratio (PSNR) will be used to evaluate the output of both the binary and stochastic Roberts edge detection designs by comparing each resulting image to the ideal (trojan free) edge detection output.
- Success Threshold:

$$PSNR_{stochastic} > PSNR_{binary}$$

Technical Constraints

A constraint is that stochastic bitstream lengths must remain within the practical limits for real hardware implementations. Although longer bitstreams increase computation accuracy, they also introduce greater latency and higher power consumption. This constraint ensures that the simulated designs remain realistic and comparable to potential implementations. Another constraint is that some hardware trojan attacks exist at the transistor or circuit level, but since we do not have access to physical hardware, our focus is instead on attacks that target high-level logic architectures that aim to reduce the accuracy or correctness of operations.

Standards

[1]"A Hardware Bill of Materials (HBOM) Framework for Supply Chain Risk Management." Available: https://www.cisa.gov/sites/default/files/2023-09/A%20Hardware%20Bill%20of%20Materials%20Framework%20for%20Supply%20Chain%20Risk%20Management%20%28508%29.pdf

This document highlights a repeatable process by which sellers can communicate all of the hardware components of their equipment. Although we don't plan to sell our SC design, we aim to keep track of all of the hardware components of our design and communicate them as instructed by this document.

[2]NIST, "NIST Special Publication 800-63B," *Nist.gov*, Oct. 16, 2023. https://pages.nist.gov/800-63-3/sp800-63b.html

The National Institute of Standards and Technology (NIST) provides widely used cybersecurity and system protection standards that apply to the software environment used in this project. NIST SP 800-63B establishes multi-factor authentication guidelines and provides guidelines on access control. Following these NIST standards ensures that the software tools, datasets, and results obtained during our project remain protected from unauthorized access or modification.

Approach & Design Methodology

We assume that a hardware trojan is inserted at the gate level of our design for both the traditional binary implementation we are comparing against and the proposed stochastic computing solution. Rather than

attempting to detect the effects of a trojan or design redundant architecture, we designed a system that produces minimal accuracy degradation when a trojan is introduced while still maintaining a high accuracy in normal operation without the need for redundant architecture. Our hardware design swaps out binary computation and number representation for stochastic operations and bitstream representations of numbers to create an implementation that resists hardware trojans applied to the outputs of operations.

Design Introduction

We combined stochastic computing and traditional Roberts edge detection algorithms to develop an edge detection hardware implementation capable of performing at a high level of accuracy with and without the presence of gate-level hardware trojan attacks. At the core of our architecture was stochastic bit stream generation. We used Sobol random number generation, a quasi-random technique to convert the pixel gradient values of images into stochastic representations. We then used an iterative approach to develop our stochastic computing architecture starting with the design, simulation, and analysis of basic stochastic operations in python and moving into a SystemVerilog implementation capable of computing all of the gradient values given an image matrix. We visualized the output image matrix in MATLAB to verify our results.

Chosen Design

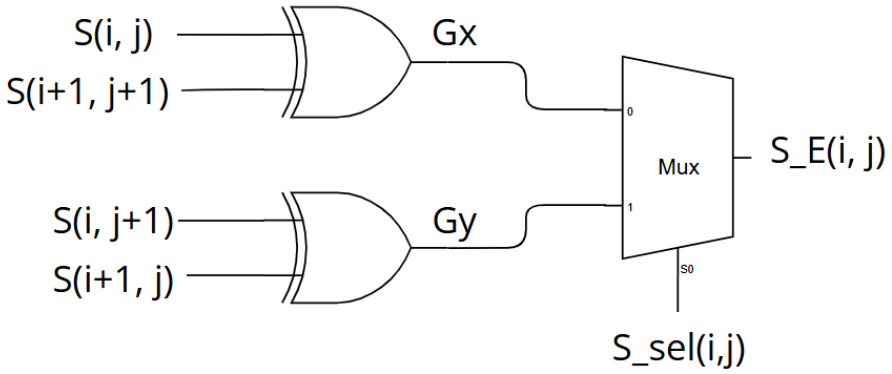
We implemented stochastic Roberts Edge detection by encoding each grayscale pixel as a stochastic bit stream in which the density of ones represented the intensity. For each pixel (i, j) we applied the Roberts 2x2 operator in the stochastic domain. We form the two diagonal gradients using bitwise XOR:

$$G_x = S_I(i,j) \oplus S_I(i+1,j+1), \qquad G_y = S_I(i,j+1) \oplus S_I(i+1,j)$$

Then, a stochastic select stream S_sel(i,j) chooses the output. This select stream represents 0.5 and thus performs scaled addition on the two outputs Gx and Gy.

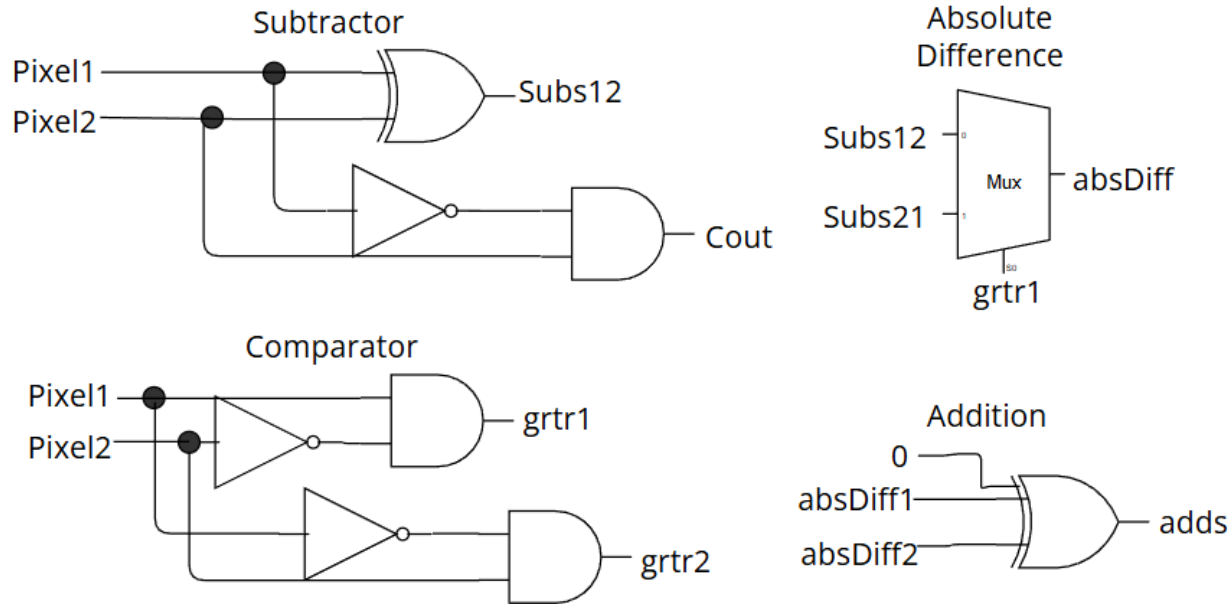
$$S_E(i,j) = \begin{cases} G_x, & S_{\text{sel}} = 1, \\ G_y, & S_{\text{sel}} = 0. \end{cases}$$

At the gate level, the core of our design is implemented with 2 XOR gates and a MUX that our stochastic number representations are fed into as shown below. After N bits (in our case we ran trials using 64 bit streams), the edge magnitude was computed by finding the density of 1s in the output stream S_E(i, j). This operation is performed on each pixel and produces a full edge map while keeping all operations in the stochastic domain.



Binary Equivalent Design

In order to test our implementation's accuracy against the traditional binary implementation, we designed a gate level Roberts edge detection circuit using 8-bit binary operations. The binary implementation is much more complex and uses more gates, but can be simplified into the following figure.



For each pixel (i, j), the Roberts operator computes two diagonal intensity differences using a 2x2 window.

$$G_x = \left|I(i,j) - I(i+1,j+1)\right|, \qquad G_y = \left|I(i+1,j) - I(i,j+1)\right|$$

Each subtraction is implemented using an 8-bit ripple-borrow subtractor and the absolute value is obtained by comparing the operands and selecting (Ia − Ib) or (Ib − Ia) accordingly. The final edge magnitude is the sum of the absolute differences:

$$E(i,j) = G_x + G_y$$

This yields the classical Roberts gradient magnitude using full binary arithmetic (bitwise subtractors, comparators, and an 8-bit ripple-carry adder.

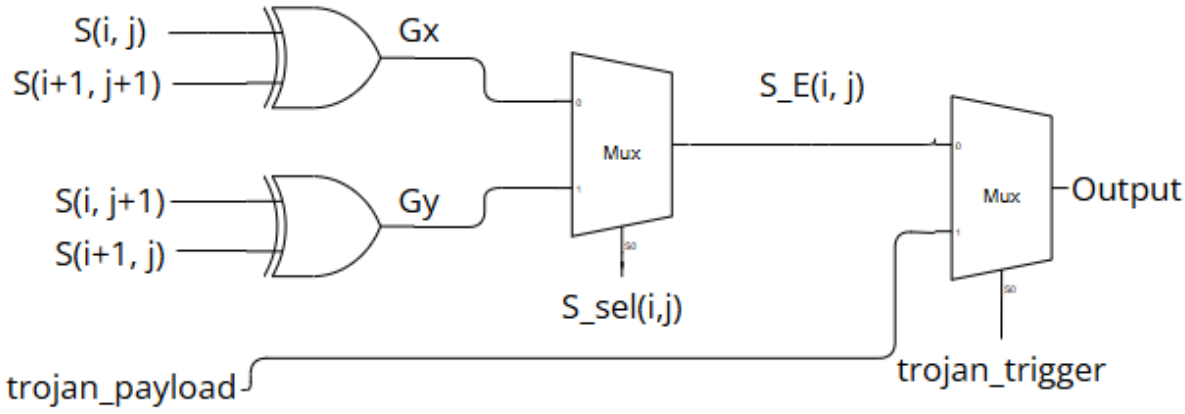
Hardware Trojan Injection

We next applied a hardware trojan to the final addition operation of both designs. This hardware trojan was implemented using a 2x1 MUX where the normal output and a trojan payload were inputs and a trojan_trigger was the select bit. The trojan_trigger select bit was based on the column number of the pixel we were calculating gradient for, and the payload was set by the row. We tested the same frequency of trojan value application on both the binary and stochastic circuits, activating the trojan for around 200 columns in each image and using a trojan payload on ⅛ of the bits in those affected pixels.

Payloads

1. Force_0: this payload forced the bit appended to the output bit stream to be a 0
2. Force_1: this payload forced the bit appended to the output bit stream to be a 1
3. Flip: this payload flipped the bit of the input and appended the flipped value to the output stream
4. Const: this payload appended the corresponding bit of a preset bit stream to the output stream

Stochastic Circuit with Trojan



Verification & Results

Visual Verification

Fig 1: Original Image

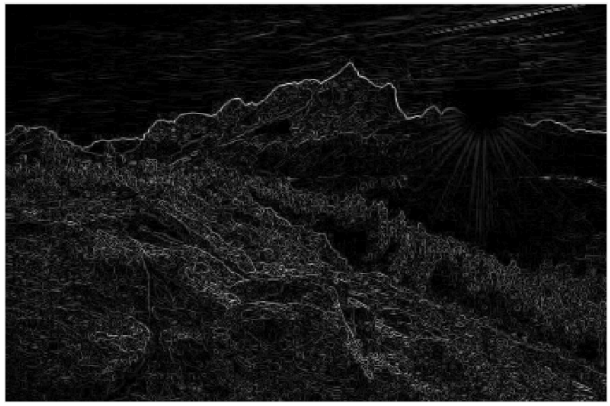


Fig 2: Ideal Edge Detecting Image

Shown above, are the original image and an ideal edge detecting image processed by simple arithmetic operations in MATLAB.

Fig 3: Binary Edge Detecting Images with no HTAs and with different HTA modes

Fig 4: Stochastic Edge Detecting Images with no HTAs and with different HTA modes

The two figures above show how binary and stochastic edge detecting images look without any HTAs and then with different HTA modes. As we can see, the binary images become more distorted by HTAs than the stochastic ones do.

Quantitative Verification

We evaluated the effectiveness of our different implementations of the Roberts Edge Detector by measuring average accuracy and peak signal-to-noise ratio (PSNR). To calculate the average accuracy, we found the error in the intensity of each individual pixel between two images, found the average of the errors, and subtracted that average from 100%. PSNR is a measure of the quality of the reconstruction of an image calculated by finding the ratio between the maximum power of the original image and the power of the noise in the reconstructed image. A higher PSNR value correspond to better quality of a reconstructed image and a PSNR value of between 30 and 50 is considered typical for lossy image and video compression.

First, we evaluated how the stochastic and binary edge detecting images were distorted by HTAs by finding the accuracy and PSNR of the images altered by HTAs compared to the images without any HTAs.

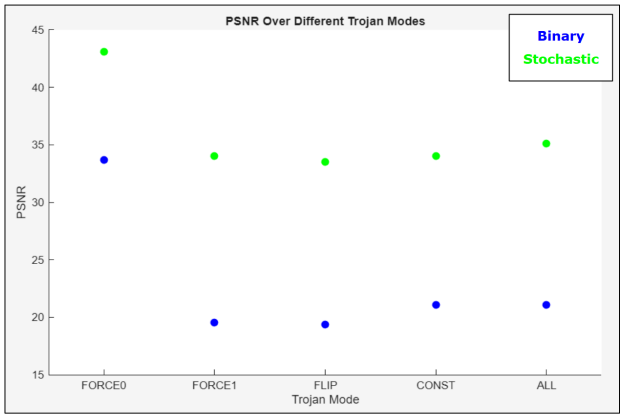


Fig 5: PSNR with and without HTAs

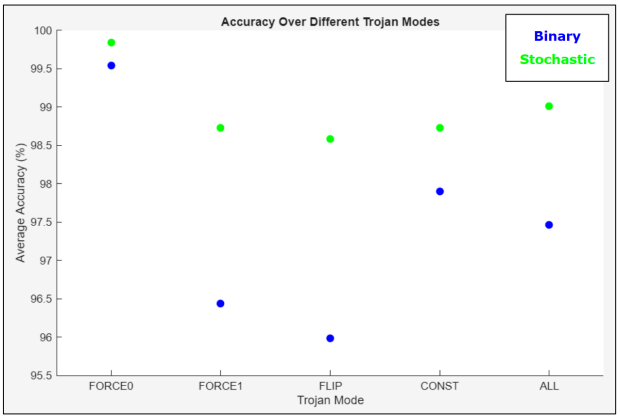


Fig 6: Accuracy with and without HTAs

As shown in the graphs, the stochastic edge detecting images maintain a higher PSNR and accuracy while facing HTAs than the binary ones do.

Next, we compared the quality of the stochastic and binary edge detecting images to that of the ideal edge detecting image.

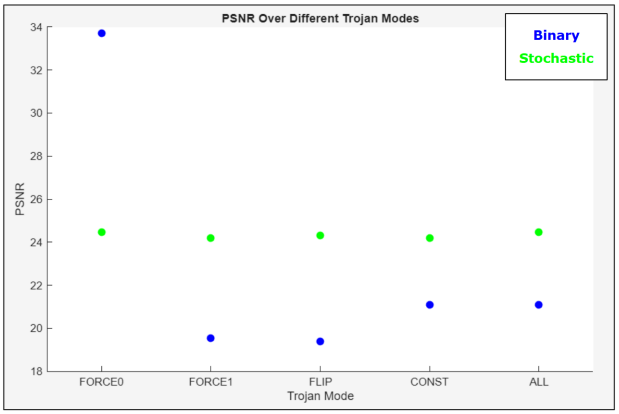


Fig 7: PSNR vs Ideal

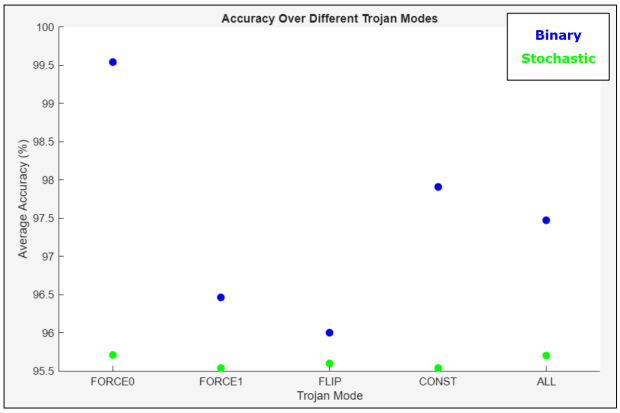


Fig 8: Accuracy vs Ideal

Compared to the ideal edge detecting image, the stochastic edge detecting images had a higher PSNR but the binary edge detecting images had a greater accuracy.

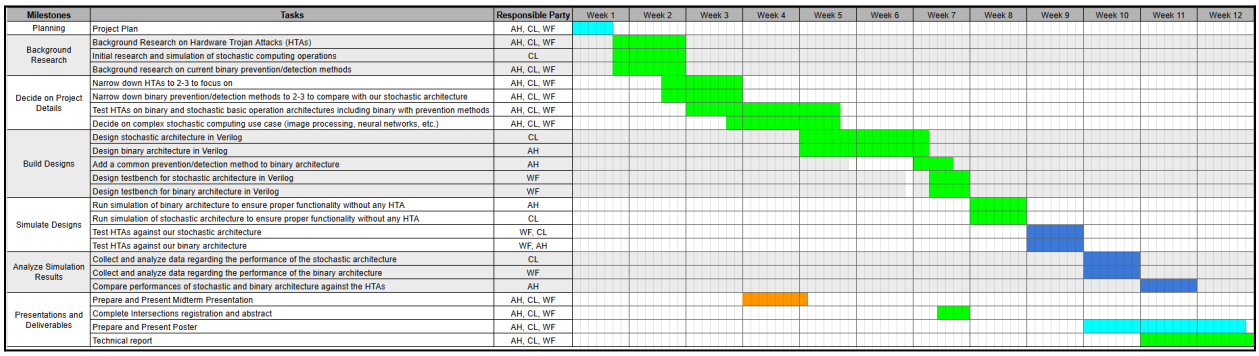
Although the binary edge detecting images showed a higher overall accuracy, the stochastic edge detecting images showed to be more resilient against HTAs which is more significant for our goals.

The PSNR and accuracy data is summarized in the table below.

	Trojan vs None				vs Ideal			
	Binary		Stochastic		Binary		Stochastic	
	PSNR	Acc. (%)	PSNR	Acc. (%)	PSNR	Acc. (%)	PSNR	Acc. (%)
No Trojan	N/A	N/A	N/A	N/A	Inf	100	24.3	95.6
FORCE0	33.7	99.5	43.1	99.8	33.7	99.5	24.5	95.7
FORCE1	19.5	96.4	34.0	98.7	19.5	96.5	24.2	95.5
FLIP	19.3	96.0	33.5	98.6	19.4	96.0	24.3	95.6
CONST	21.1	97.9	34.0	98.7	21.1	97.9	24.2	95.5
ALL	21.1	97.5	35.1	99.0	21.1	97.5	24.5	95.7

Project Management

Gantt Chart:



Milestones	Tasks	Responsible Party	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12
Planning	Project Plan	AH, CL, WF												
Background Research	Background Research on Hardware Trojan Attacks (HTAs)	AH, CL, WF												
	Initial research and simulation of stochastic computing operations	CL												
	Background research on current binary prevention/detection methods	AH, CL, WF												
Decide on Project Details	Narrow down HTAs to 2-3 to focus on	AH, CL, WF												
	Narrow down binary prevention/detection methods to 2-3 to compare with our stochastic architecture	AH, CL, WF												
	Test HTAs on binary and stochastic basic operation architectures including binary with prevention methods	AH, CL, WF												
	Decide on complex stochastic computing use case (image processing, neural networks, etc.)	AH, CL, WF												
Build Designs	Design stochastic architecture in Verilog	CL												
	Design binary architecture in Verilog	AH												
	Add a common prevention/detection method to binary architecture	AH												
	Design testbench for stochastic architecture in Verilog	WF												
	Design testbench for binary architecture in Verilog	WF												
Simulate Designs	Run simulation of binary architecture to ensure proper functionality without any HTA	AH												
	Run simulation of stochastic architecture to ensure proper functionality without any HTA	CL												
	Test HTAs against our stochastic architecture	WF, CL												
	Test HTAs against our binary architecture	WF, AH												
Analyze Simulation Results	Collect and analyze data regarding the performance of the stochastic architecture	CL												
	Collect and analyze data regarding the performance of the binary architecture	WF												
	Compare performances of stochastic and binary architecture against the HTAs	AH												
Presentations and Deliverables	Prepare and Present Midterm Presentation	AH, CL, WF												
	Complete Intersections registration and abstract	AH, CL, WF												
	Prepare and Present Poster	AH, CL, WF												
	Technical report	AH, CL, WF												

Relevant Courses

Andrew

Course	Description
ECSE281	• Verilog • ModelSim
ECSE313	• Image processing in MATLAB

Camden

Course	Description
ECSE 281	● Combinational Logic Circuits
ECSE 318	● VLSI ● Verilog
CSDS 133/313	● Python ● Data Visualization
ECSE 315	● Digital Systems Design

William

Course	Description
ECSE 281	● Verilog ● ModelSim
ECSE 313	● Image Processing in MATLAB
CSDS 313	● Data analysis and visualization

Appendix A

```
/*// Perform Roberts Edge Detection on the image matrix and fills edges
matrix with results
for (int i = 0; i < 293; i++) begin
  for (int j = 0; j < 439; j++) begin
    reg [63:0] gradx, grady;
    reg [63:0] normal_edge;
    int        payload_mode;
    bit        trojan_active;

    // Normal Roberts computation
    gradx       = image[i][j]   ^ image[i+1][j+1];
    grady       = image[i][j+1] ^ image[i+1][j];
    normal_edge = selstream[i][j] ? gradx : grady;

    // Trigger: Trojan active in specified column range
    trojan_active = (j >= 150) && (j < 350);

    // Payload selection based on i
    if (i < 73)
      payload_mode = MODE_FORCE_0;
    else if (i < 146)
      payload_mode = MODE_FORCE_1;
    else if (i < 219)
      payload_mode = MODE_FLIP;
    else
```

```
  payload_mode = MODE_CONST;


// To use a single payload mode for all bits
payload_mode = MODE_FLIP;


// Apply Trojan at the output on specified bits
if (!trojan_active) begin
  edges[i][j] = normal_edge;
end
else begin
  case (payload_mode)
    // Force 0 only on masked bits
    MODE_FORCE_0: edges[i][j] =
      normal_edge & ~BIT_MASK;


    // Force 1 only on masked bits
    MODE_FORCE_1: edges[i][j] =
      normal_edge | BIT_MASK;


    // Flip only masked bits
    MODE_FLIP:    edges[i][j] =
      normal_edge ^ BIT_MASK;


    // Replace masked bits with Trojan constant bits
    MODE_CONST:   edges[i][j] =
      (normal_edge & ~BIT_MASK) | (trojan_const & BIT_MASK);
```

```
        default:      edges[i][j] = normal_edge;
      endcase
    end
  end
end
```

/* code 1

```
// Perform Roberts Edge Detection on the image matrix and fills edges
matrix with results
for(int i = 0; i < 293; i++)
begin
    for(int j = 0; j < 439; j++)
    begin
        //Absolute Difference |image[i][j] - image[i+1][j+1]|
        //image[i][j] - image[i+1][j+1]
        subsAB[0] = image[i][j][0] ^ image[i+1][j+1][0];
        CoutsSubAB[0] = ~image[i][j][0] & image[i+1][j+1][0];
        for(int k = 1; k <= 7; k++)
        begin
            subsAB[k] = (image[i][j][k] ^ image[i+1][j+1][k]) ^
CoutsSubAB[k-1];
            CoutsSubAB[k] = (~image[i][j][k] & image[i+1][j+1][k]) |
((~image[i][j][k] ^ image[i+1][j+1][k]) & (CoutsSubAB[k-1]));
        end
        //image[i+1][j+1] - image[i][j]
        subsBA[0] = image[i+1][j+1][0] ^ image[i][j][0];
        CoutsSubBA[0] = ~image[i+1][j+1][0] & image[i][j][0];
        for(int k = 1; k <= 7; k++)
```

```
        begin
            subsBA[k] = (image[i+1][j+1][k] ^ image[i][j][k]) ^
CoutsSubBA[k-1];
            CoutsSubBA[k] = (~image[i+1][j+1][k] & image[i][j][k]) |
((~image[i+1][j+1][k] ^ image[i][j][k]) & (CoutsSubBA[k-1]));
        end
        //Comparator
        grtrA[7] = image[i][j][7] & ~image[i+1][j+1][7];
        grtrB[7] = ~image[i][j][7] & image[i+1][j+1][7];
        for(int k = 6; k >= 0; k--)
        begin
            grtrA[k] = grtrA[k+1] | ~grtrB[k+1] & (image[i][j][k] &
~image[i+1][j+1][k]);
            grtrB[k] = grtrB[k+1] | ~grtrA[k+1] & (~image[i][j][k] &
image[i+1][j+1][k]);
        end
        //Final Difference
        for(int k = 0; k <= 7; k++)
        begin
            absDiff[k] = (subsAB[k] & grtrA[0]) | (subsBA[k] &
~grtrA[0]);
        end


        //Absolute Difference |image[i+1][j] - image[j][j+1]|
        //image[i+1][j] - image[i][j+1]
        subsABx[0] = image[i+1][j][0] ^ image[i][j+1][0];
        CoutsSubABx[0] = ~image[i+1][j][0] & image[i][j+1][0];
        for(int k = 1; k <= 7; k++)
```

```
        begin
            subsABx[k] = (image[i+1][j][k] ^ image[i][j+1][k]) ^
CoutsSubABx[k-1];
            CoutsSubABx[k] = (~image[i+1][j][k] & image[i][j+1][k]) |
((~image[i+1][j][k] ^ image[i][j+1][k]) & CoutsSubABx[k-1]);
        end
        //image[i][j+1] - image[i+1][j]
        subsBAx[0] = image[i][j+1][0] ^ image[i+1][j][0];
        CoutsSubBAx[0] = ~image[i][j+1][0] & image[i+1][j][0];
        for(int k = 1; k <= 7; k++)
        begin
            subsBAx[k] = (image[i][j+1][k] ^ image[i+1][j][k]) ^
CoutsSubBAx[k-1];
            CoutsSubBAx[k] = (~image[i][j+1][k] & image[i+1][j][k]) |
((~image[i][j+1][k] ^ image[i+1][j][k]) & CoutsSubBAx[k-1]);
        end
        //Comparator
        grtrAx[7] = image[i+1][j][7] & ~image[i][j+1][7];
        grtrBx[7] = ~image[i+1][j][7] & image[i][j+1][7];
        for(int k = 6; k >= 0; k--)
        begin
            grtrAx[k] = grtrAx[k+1] | (~grtrBx[k+1] & (image[i+1][j][k]
& ~image[i][j+1][k]));
            grtrBx[k] = grtrBx[k+1] | (~grtrAx[k+1] & (~image[i+1][j][k]
& image[i][j+1][k]));
        end
        //Final Difference
        for(int k = 0; k <= 7; k++)
```

```
        begin
            absDiffx[k] = (subsABx[k] & grtrAx[0]) | (subsBAx[k] &
~grtrAx[0]);
        end

        //Addition
        adds[0] = absDiff[0] ^ absDiffx[0] ^ 1'b0;
        CoutsAdd[0] = (absDiff[0] & absDiffx[0]) | (1'b0 & (absDiff[0] ^
absDiffx[0]));
        for(int k = 1; k <= 7; k++)
        begin
            adds[k] = absDiff[k] ^ absDiffx[k] ^ CoutsAdd[k-1];
            CoutsAdd[k] = (absDiff[k] & absDiffx[k]) | (CoutsAdd[k-1] &
(absDiff[k] ^ absDiffx[k]));
        end

        // Hardware Trojan on the adder output
        reg       trojan_active;
        integer   payload_mode;
        reg [7:0] normal_edge;
        reg [7:0] bit_mask;      // dynamic mask
        integer   bit_index;     // which bit to hit 0..7

        normal_edge = adds;  // "honest" Roberts magnitude

        // Sequential trigger: Trojan only active in certain columns
        trojan_active = (j >= TROJAN_COL_START) && (j < TROJAN_COL_END);
```

```
        // Choose which bit position to corrupt for this pixel based on
row
        bit_index = (i * j) % 8;

        // Create a mask with a 1 at bit_index, 0 elsewhere
        bit_mask = 8'b0000_0001 << bit_index;

        // Payload selection based on row i (four horizontal bands)
        if (i < 73)
            payload_mode = MODE_FORCE_0;
        else if (i < 146)
            payload_mode = MODE_FORCE_1;
        else if (i < 219)
            payload_mode = MODE_FLIP;
        else
            payload_mode = MODE_CONST;

        // Apply Trojan: only modify the bit chosen by bit_mask
        if (!trojan_active) begin
            edges[i][j] = normal_edge;
        end
        else begin
            case (payload_mode)
                MODE_FORCE_0: edges[i][j] =
                    normal_edge & ~bit_mask;                    // force
that bit to 0
```

```
                MODE_FORCE_1: edges[i][j] =
                    normal_edge | bit_mask;                    // force
that bit to 1

                MODE_FLIP:    edges[i][j] =
                    normal_edge ^ bit_mask;                    // flip
that bit

                MODE_CONST:   edges[i][j] =
                    (normal_edge & ~bit_mask) | (TROJAN_CONST &
bit_mask);
                                                                // take
that bit from TROJAN_CONST
                default:      edges[i][j] = normal_edge;
            endcase
        end
    end
end
```

Code 2: