*A. Overview*

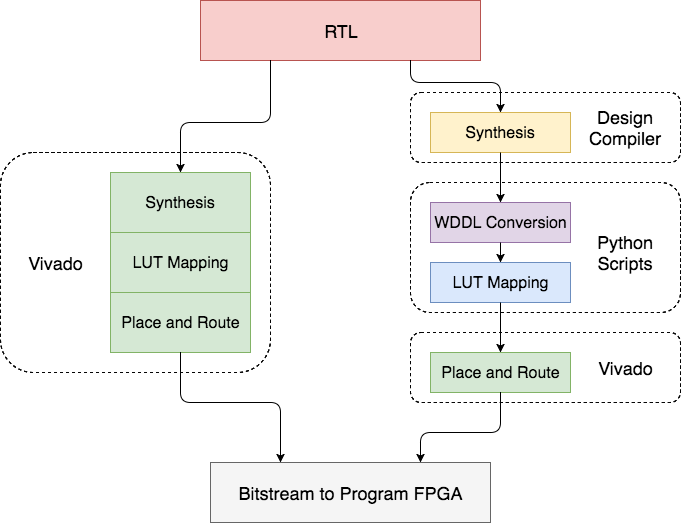
Implementations of AES on FPGAs are vulnerable to power analysis attacks. In my research I worked to implement an AES core that is resistant to power analysis. I focused on Correlation Power Attacks (CPA) attacks because, as talked about before, it is the most effective in deciphering specific bit values from a collection of power traces. Attackers conducting a CPA attack on a device running AES have the ability to get around the AES algorithm without using cryptanalysis. They can use CPA to decipher keys from the device and once these keys are uncovered the system is compromised.

To solve this problem, my research was aimed at implementing a CPA resistant gate-level AES-128 Verilog design onto an FPGA. This implementation was a gate-level Wave Dynamic Differential Logic (WDDL) protected AES-128 Verilog design and it was programed onto the Artix 7 FPGA located on the ChipWhisperer CW305 target board. To test the security of the design implementation, I used the ChipWhisperer platform. This platform makes it easy to record power traces from the FPGA and then the ChipWhisperer software use the traces to conduct a CPA attack on the AES core. From there I used the number of traces needed to successfully uncover the keys as the metric to compare the security of each design. Along with this, I recorded the number of LUTs needed to implement each design to examine how the area of the design increases as different stages of the algorithm are protected.

ChipWhisperer is an open source toolchain that makes it easy for a user to perform side channel attacks on specified target boards. It consists mainly of two software programs, the ChipWhisperer Capture program and the ChipWhisperer Analyzer program. Along with this, it consists of two types of boards, one is the target board, used for performing the encryption operation, and the other is the capture board, used for capturing power traces from the target board [2]. The capture board I used in my research is the CW1173 ChipWhisperer-Lite and the target board I used is the CW305 Artix FPGA Target. The encryption algorithm ran on the CW305 and the power traces were captured by the CW1173.

I compared several types of AES Verilog designs to see how their security and area vary. The baseline was a gate-level AES core with no additional defences added to it. Next, I implemented an AES core with WDDL protection, minus the precharge phase and LUT instantiation of each gate, on each individual stage of the AES algorithm. WDDL was first implemented on the SubBytes stage, then the MixColumns stage, and finally the AddRoundKey stage. I also tested various combinations of these stages protected by WDDL. The next step was to implement a WDDL protected AES core with LUT instantiation of each gate and repeat the attacks as done previously. Lastly, from there I implemented a precharge phase and repeated the attacks. All of these implementations will be attacked using the ChipWhisperer platform and the resulting number of power traces needed to uncover the keys will be recorded.

Figure 11 shows the overall flow of my research. Vivado is a software, developed by Xilinx, that synthesizes and analyzes Verilog designs. When provided with an RTL design, Vivado synthesizes it, maps the gates to LUTs, and then executes place and route. The result is a bitstream that is used to program the FPGA. In my research I had to stray from this approach because Vivado tries to optimize the design before generating a bitstream. This means, various gates are mapped to various LUTs on the FPGA, which in our case is counteractive to balancing load capacitances of the gates in the design.



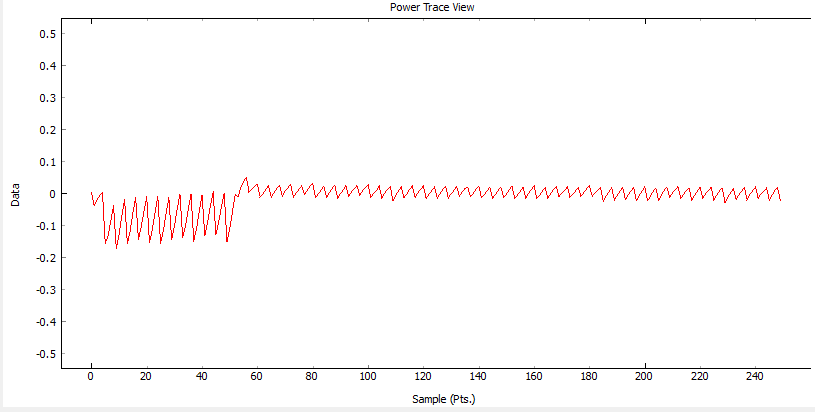
**Figure 11:** Overview of research components starting with RTL design all the way to the bitstream that programs the FPGA. The left side allows Vivado to perform all the LUT mapping and Place and Route. The right side, my approach, handles the mapping of gates to LUTs to avoid any optimization that may be done by Vivado. This ensures the load capacitance balancing for each gate.

My approach for generating a bitstream started with synthesising an AES-128 design, with the help of one of Prof. Holcomb’s PHD students, Siva Nishok Dhanuskodi, to its gate-level Verilog version using a design compiler. From there the gate-level AES-128 Verilog design was modified using Python scripts to protect the design with WDDL format and to force the instantiation of each gate to its own LUT. The last step was to use Vivado to execute the remaining place and route and then generating a bit stream to program the FPGA.

*B. ChipWhisperer Basic DPA Attack on AES*

The first step in my research was to learn how to conduct a DPA attack using the ChipWhisperer platform. To do this I started with running through some basic tutorials that covered capturing power traces from a target board using the ChipWhisperer Capture board then using traces to conduct attack.

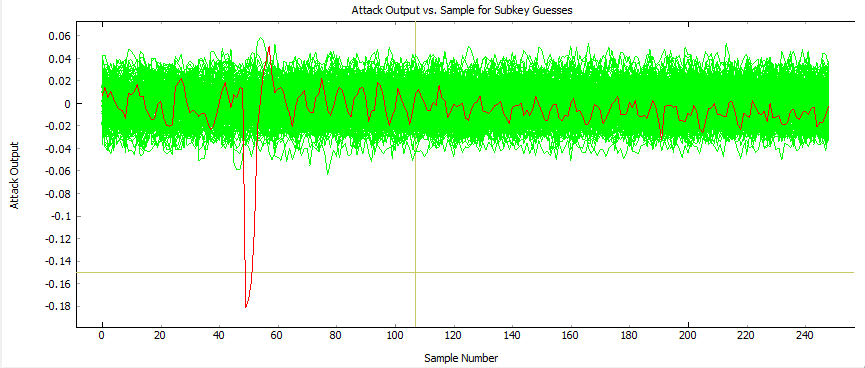
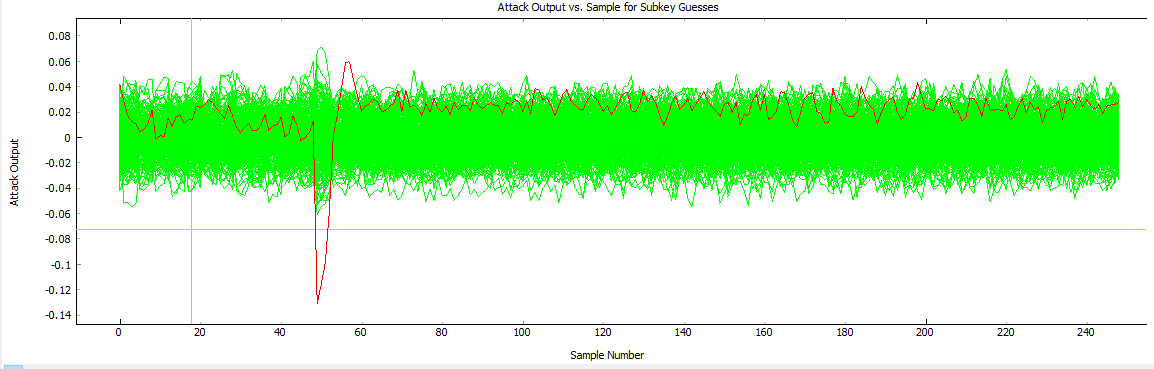
The capturing of power traces from the ChipWhisperer CW305 target board is done by the following steps. First, the hardware needed to be correctly connected. This means the target board needed to be connected to the capture board and both of these boards needed to be connected to the PC running the ChipWhisperer software. From there, to capture the power traces, the ChipWhisperer Capture application needed to be used. I used the bitstream provided by ChipWhisperer, which contained a default AES core, to program the Artix 7 FPGA located on the CW305 target board. The capture program, when told which capture and target board it was connecting to, ran the encryption algorithm on the Artix 7 FPGA a number of times and captured the resulting power traces of the device. Figure 12 shows a single power trace of an encryption run. The first set of eleven spikes from sample points 0 to 50 represent the eleven stages of encryption. For the purpose of this tutorial the number of power traces recorded was 5000. It took just over 10 minutes to capture all the traces.



**Figure 12:** Power trace of one run of the AES algorithm. The power spikes that start the power trace represent the eleven rounds of the AES algorithm.

The next step involved importing the power traces that were collected into the ChipWhisperer Analyze program. The Analyze program takes in all the power traces and uses them to conduct a CPA attack. I set the target point to the output of the S-Box in the last round of encryption. The attack took about 9 minutes and the keys of the system were uncovered. Considering this attack in full only took about 20 minutes, the default AES core used in the tutorial is not a secure one against side channel attacks.

The following figure shows the attack output vs the sample for subkey guesses for byte 0 and byte 7 of the round key. The correct key guess is highlighted in red and the incorrect key guesses are shown in green. As you can see, around sample number 50 there is a spike in the attack output trace. This sample number correlates to the last round of the encryption algorithm where the attack was targeting.



**Figure 13:** The spike around sample number 50 represents a correct key guess for both byte 0, shown in top figure, and byte 7, shown in bottom figure. The spikes occur at the same point in each plot because this is the point in the power trace where the output of the S-Box is evaluated and where the attack is targeting. The incorrect key guesses hover around 0 with no prevalent spikes.

*C. WDDL Conversion Script*

The next step in my research was to start implementing some side channel defences to enhance the security of the AES core. The goal was to incrementally increase the amount of traces needed to uncover the subkeys of the system. The AES core provided by ChipWhisperer was not a gate-level design. To implement WDDL, which was my first defence to implement, I needed to be able to work with a gate-level AES core consisting of only AND and OR gates.

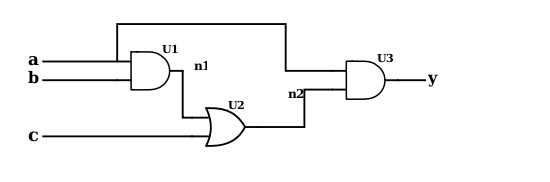
Prof. Holcomb’s PHD student, Siva Nishok Dhanuskodi, who also works in Knowles Engineering Building room 314, was able to assist me in synthesizing an AES core down to gate-level Verilog. He used Synopsys Design Compiler to map a Verilog AES core to a gate-level version of the Verilog AES core.

Once I had the gate-level AES design, the next step was to write a Python script to parse this file. The main goal of this script was to be able to convert specified modules in the AES core to WDDL format. The user inputs the names of the modules that need to be converted and the script parses the AES Verilog design, modifies the appropriate modules, and then writes the resulting Verilog AES core to a new file.

I found it easiest for the script to read in the entire Verilog file first and organize it based on the different modules. I created a Module object for each module in the Verilog file. This Module object contained all the inputs, outputs, wires, and gates contained in the module.

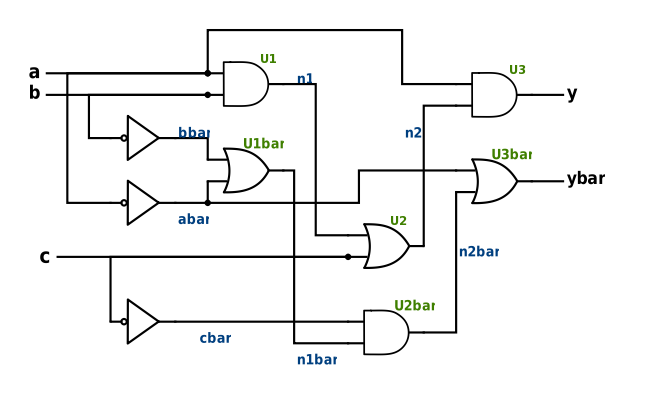
Once the entire Verilog file was parsed and all the modules were recorded in Module object list, the next step was to write the modules back to the output file and at the same time modify any of the modules that needed to be converted to WDDL. If the module was to be converted to WDDL format the following occurs. For every input and output in the module, a wire needed to be declared to hold the complement of that input or output. Along with this, for every wire declared in the module there needed to be an additional wire declared to hold the complement value of the wire. The next step was to modify the gate list. This is where the real WDDL modification took place. The script iterates through the gate list and for every AND or OR gate, it took its inputs and its output, found their complements, and created a complement gate for it. This complement gate was stored back into the gate list to be written to the new file.

The development of this script was done in two stages. The first stage involved working with a simple Verilog design, shown below, consisting of only a couple inputs and outputs and multiple AND and OR gates. This circuit was used for debugging purposes. Once the script was developed and tested on this simple design, I switched to working with the gate-level AES core, and was able to successfully produce an AES core with WDDL implemented on the selected modules.



|  |
| --- |
| module sample\_circuit ( a, b, c, y );  input a, b, c;  //input\_done  output y;  //output\_done  wire n1, n2;  //wire\_done  AND2\_X1 U1 ( .A1(b), .A2(a), .ZN(n1) );  OR2\_X1 U2 ( .A1(c), .A2(n1), .ZN(n2) );  AND2\_X1 U3 ( .A1(a), .A2(n2), .ZN(y) );  endmodule  //done |

**Figure 14:** This is the simple circuit used when creating the Python Script. The Verilog code for the design is also shown.



|  |
| --- |
| module sample\_circuit ( a, b, c, y );  input a;  input b;  input c;  wire abar;  wire bbar;  wire cbar;  assign abar = ~a;  assign bbar = ~b;  assign cbar = ~c;  //input\_done    output y;  wire ybar;  //output\_done    wire n1;  wire n2;  wire n1bar;  wire n2bar;  //wire\_done    AND2\_X1 U1 ( .A1(b), .A2(a), .ZN(n1) );  OR2\_X1 U1bar ( .A1(bbar), .A2(abar), .ZN(n1bar) );  OR2\_X1 U4 ( .A1(c), .A2(n1), .ZN(n2) );  AND2\_X1 U4bar ( .A1(cbar), .A2(n1bar), .ZN(n2bar) );  AND2\_X1 U4 ( .A1(a), .A2(n2), .ZN(y) );  OR2\_X1 U4bar ( .A1(abar), .A2(n2bar), .ZN(ybar) );  endmodule  //done |

**Figure 15:** The script converted the test circuit into its WDDL format. Wires were added to hold the complement values of the input, output, and previously instantiated wires. Complement gates we also added to the design.

The script takes only several seconds to execute and, because of the way it was design, it is easy for the user to be able to change the modules that need to be modified. This allowed me to quickly modify and generate different WDDL designs for the AES core.

*D. LUT Mapping Script*

One of the main aspects of securing this FPGA AES core, which is mentioned in the review of literature section, is balancing the load capacitances of each individual WDDL cell. To do this, the FPGA needs to be forced to instantiate every gate to its own LUT. If this is not done, the design will be optimized, multiple gates will be combined into the same LUTs, and the gate level structure of the the AES core will be dismantled.

To start, I created another Python script that instantiated each individual gate to its own lookup table. This script simply parsed the already modified AES design file, and for every AND or OR gate, it performed the Verilog modification shown below. These two Python scripts allow me to generate a WDDL gate-level AES-128 core which can be programmed on to the FPGA.

|  |
| --- |
| AND2\_X1 U4 ( .A1(n2), .A2(n3), .ZN(y) );  OR2\_X1 U4bar ( .A1(n2bar), .A2(n3bar), .ZN(ybar) ); |

|  |
| --- |
| //AND U4  LUT2 U4(  .INIT(4'h8)  ) LUT2\_inst (  .O(y)  .I0(n2)  .I1(n3)  );  //OR U4bar  LUT2 U4bar(  .INIT(4'he)  ) LUT2\_inst (  .O(ybar)  .I0(n2bar)  .I1(n3bar)  ); |

**Figure 16:** The Verilog code uses macros to force the FPGA to instantiate each gate to its own LUT. When inputted to Vivado to generate a bitstream, this macros will prevent Vivado from attempting to optimize the design.

*E. Methods of Data Collection*

One issue I had with the ChipWhisperer program was that it did not directly tell you how many power traces it needed to successfully uncovered the keys. The way the data outputs were set up was that the ChipWhisperer Analyzer program would take in all the power traces recorded, iterate through all of them during the attack, and the results would be updated in the results table, shown in figure ??, as the attack went on. This was an issue because the attack would not stop when the correct key was found, it continued through all the power traces provided. For me to figure out how many power traces it took to uncover the keys, one would have to be watching the results table throughout the entire attack which was not possible for an attack that may last several hours.

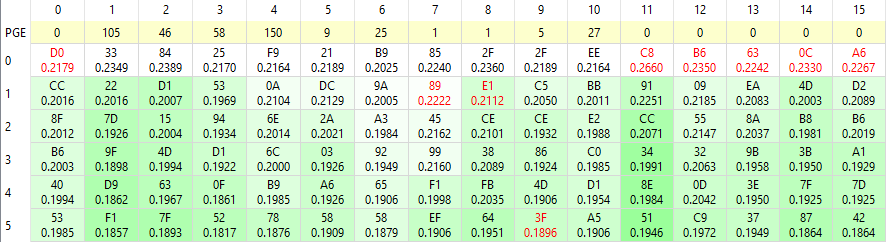


Figure ??: Results table of a partially completed attack. The bytes highlighted in red are the correct key byte. These correct key bytes slowly make their way to the top of the table as more power traces are examined and the correlation of the incorrect key guesses depreciates.

To get around this issue I had to go into the application code to find where the result table values were updated throughout an attack. After finding the correct file I made the following additions in figure ?? and ??. These additions took the values that were going to be written to the results table and also wrote them to a text file. In the application code I added a conditional that check to see if the correct subkey guess was in the top spot of the results table. From there I checked to see if all 16 subkeys were in the top spots. Once this was done I set a threshold saying, if these correct subkeys all stay in the same top spot for 200 power traces in a row then the correct subkey had been successfully uncovered. Once this has happened, the total number of power traces needed to successfully uncover the correct key were written to a text file.

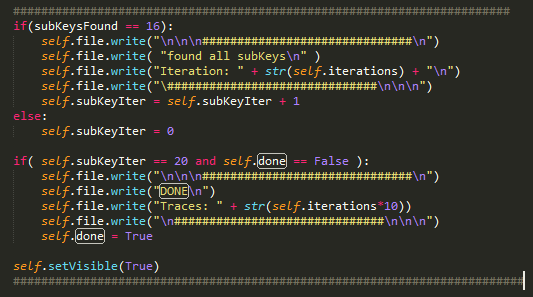


Figure ??: This modification in the code counted the number of traces in which all the correct subkeys were in the top position of the results table. It printed “DONE” to a text file when the correct key was uncovered and also printed the total number of power traces needed to get to that point.

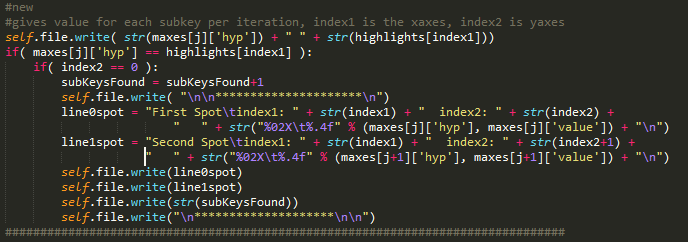
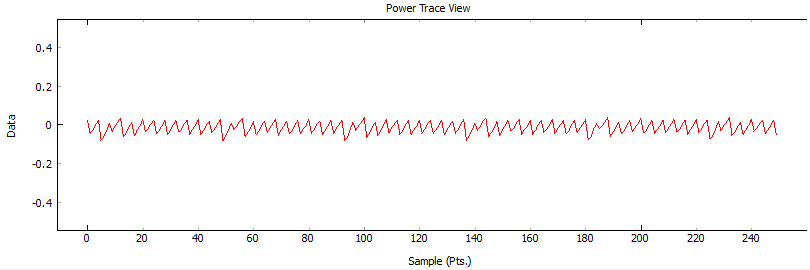


Figure ??: Here the code is examining the individual key byte guesses. This is where I made the modification to count the number of correct subkey guesses that were already at the top of the results table.

For each design, it was be attacked 5 times and the average number of power traces to uncover the key was be recorded along with the number of LUTs it took to implement the design.

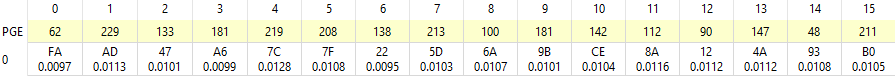
*F. Programming FPGA and Conducting DPA Attack*

It took several weeks of configurations and design modification to successfully attack the basic gate-level AES core I was using. At first I attempted to substitute the entire gate-level AES design for the default AES design previously attack in the tutorial. This substitution was relatively simple and the only modifications that needed to be made to the AES core were incorporating several signals used in the default AES design. These signals included *load\_i*, which signaled when a new value was to encrypted, and *busy\_o*, which signaled when the encryption of a piece of data was complete. When programmed on the FPGA and attacked, the following power trace was recorded.



**Figure 17:** Single power trace of implemented AES core. The eleven peaks repeat five times representing the encryption algorithm running five times.

I collected 15,000 power traces and after the attack iterated through all of them the correct key was not uncovered and the it seemed that it never would be. Although this may seem like a good sign for the design, considering this AES core was the exact same as the tutorial design, the attack should have completed relatively quickly and the correct key should have taken no more than 5,000 power traces to uncover. This told me that the power traces being compared were not lining up meaning the signals I incorporated into the design were not being triggered at the right times.



**Figure 18:** The results table shows the key guess which had the greatest correlation. ChipWhisperer highlights the correct key guess in red and since none of these guesses are highlighted in red the attack was not only unsuccessful in finding the correct key, it was unsuccessful finding even one byte of the correct key.

After trying a few different modifications and getting the same result I decided to change my approach. Instead of completely substituting the gate-level AES design for the default on I incorporated the gate-level design into the default one. This was done by taking out the sections in the default AES design which were related to the SubBytes stage, the MixColumn stage, and the AddRoundKey stage. I replaced those sections with the gate-level AES stages from the gate-level AES design. I was planning on modifying the ShiftRows section too, but after further examination modifying this sections would not make a difference because it contains no AND or OR gates. This allowed the AES core to have the appropriate signals to be attacked by the software as well as gate-level stages which can still be modified and protected by WDDL.

After substituting in sections of my design into the default design I was able to successfully attack it. Figure ?? shows a single power trace from the encryption. As you can tell, this looks very similar to the power trace recorded during the tutorial. I collected 5000 power traces using the ChipWhisperer Capture program and then used the Analyzer program to conduct the attack on the traces. After running the attack 5 times, the average number of traces needed to successfully uncover the key was 1526. Along with this, 1958 LUTs were used in this design. Because these numbers were similar to the default AES core, I was able to conclude that this was the correct design to use going forward in my research.

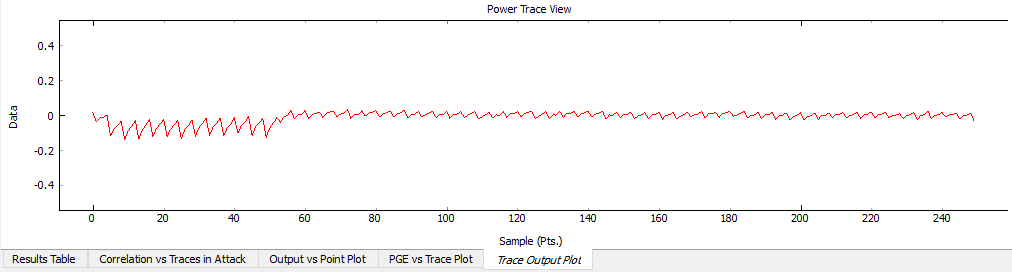


Figure ??: Single power trace of the baseline gate-level AES encryption core. This trace is very similar to the trace I got from the tutorial attack with the repeating 11 spikes at the beginning and then once the encryption ends the power trace goes flat.

*G. WDDL Without Precharge Phase*

Once the baseline attack was completed I moved to modifying the AES core to begin implementing WDDL. I started by only implementing the logic of WDDL, without the precharge phase, to observe how the addition of area can affect the security of the design. To do this I used the python scripts, discussed earlier, to modify the already existing design. I systematically begin to modify the design by first only focusing on the individual AES stages by only modifying the SubBytes portion, then the AddRoundKey portion, and finally to the MixColumn portion. From there I began to modify different combinations of stages and observed how the security of the design changed as well as how the area of the design changed.

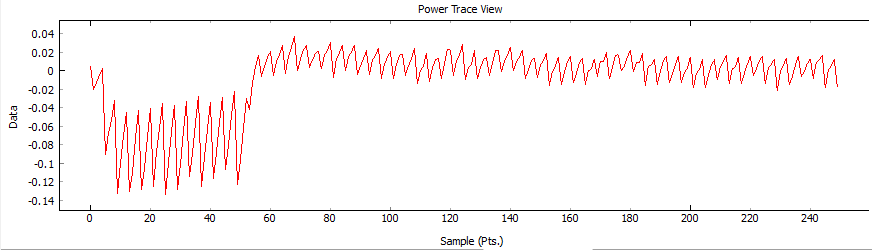


Figure ??: Power trace of WDDL protection without precharge phase and gates instantiated to their own LUTs for every stage of AES algorithm. Trace varies from baseline attack but the eleven stages of AES are still easily identifiable.

*H. WDDL With Precharge Phase*

The next step for me was to fully implement WDDL on my design by incorperating the precharge phase. To do this I had to make some slight modifications to the design. First off, I had to start with the AES\_CORE module, which was the driver for the algorithm and shown in Appendix A. To start, I modified the module so it not only takes in the the data to be encrypted but it also takes in the complement of this data. Both of these values pass through the different stages of AES. The next change I added the NORing of the inputs with the clock signal, this is how the precharge phase is implemented. This ensures that when the clock goes high, regardless of the value of the input, the NOR gate will produce a 0. Because this is happening with all the input signals, only 0’s are entering the design, which will therefore evaluate every single gate in design to a 0.

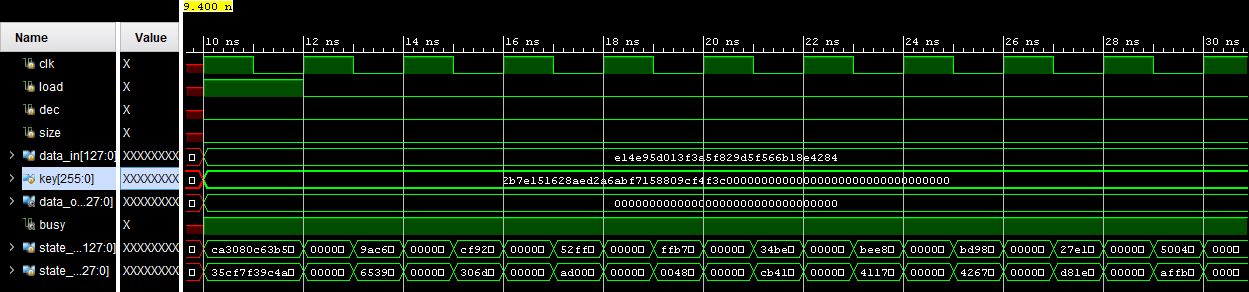


Figure ??: Simulation of AES algorithm protected by WDDL with precharge phase. The data being processed gets set to 0 on rising edge of clock and then flips to actual values on falling edge of clock.

An example of how a simple circuit is changed when WDDL with a precharge phase is implemented is shown in figure ??. The clock signal is now included in the module design as well as NOR gates.

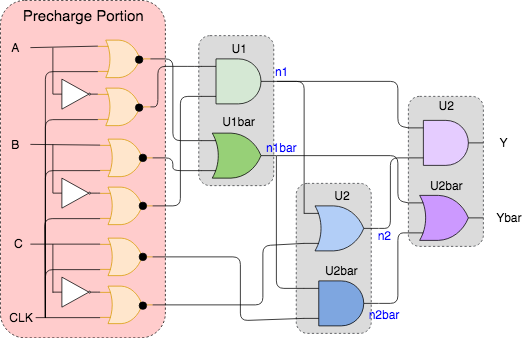
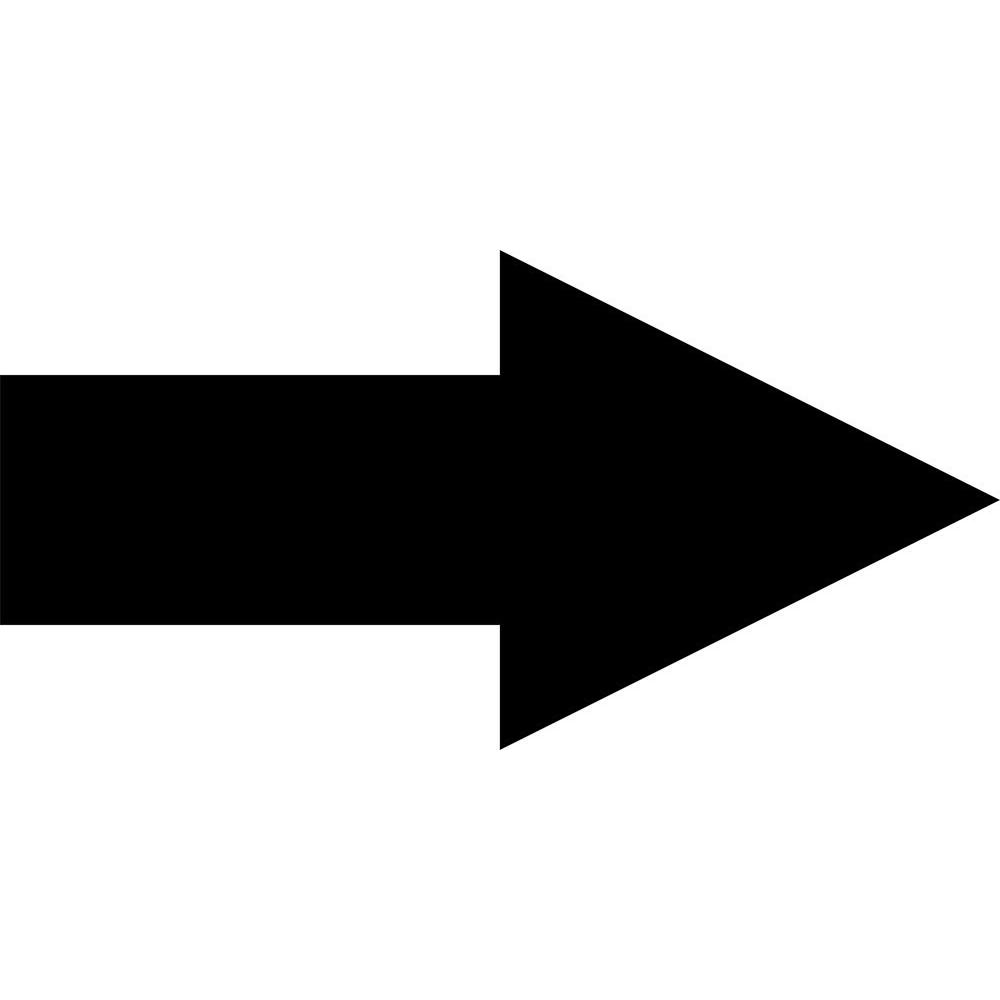
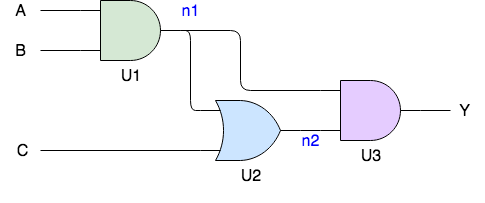


Figure ??: A fully protected circuit has gate complements and precharge phase. The clock and NOR gates are needed to implement the precharge phase.

Once these changes were made, the design was officially a WDDL protected design. From here I repeated the process as I did when working with the WDDL design without the precharge phase and recorded the area and security of each design.