EVALUATION OF SECURITY OF AES-128 WAVE DYNAMIC DIFFERENTIAL LOGIC IMPLEMENTATION ON A FIELD PROGRAMMABLE GATE ARRAY

An Honors Thesis

Presented by

Richard Hartnett

Completion Date:
May 2018

Approved By:

_____

Prof. Dan Holcomb, Electrical and Computer Engineering Department

_____

Prof. Neal Anderson, Electrical and Computer Engineering Department

ABSTRACT

Title: **Evaluation of Secuirty of AES-128 Wave Dynamic Differential Logic Implementation of Field Programmable Gate Array**
Author: **Richard Hartnett**
Thesis/Project Type: **Independent Honors Thesis**
Approved By: **Prof. Dan Holcomb, Electrical And Computer Engineering Department**
Approved By: **Prof. Neal Anderson, Electrical And Computer Engineering Department**

Embedded Systems, because of their simplicity and specific functionalities, can be vulnerable to side channel attacks. These side channel attacks are aimed not at the mathematics of an encryption algorithm, but rather the way the algorithm is implemented in hardware. This is done by using the electrical characteristics of the device to extract system information. One example of a side channel attack, and the specific focus of this paper, is a power analysis attack, which uses the total power consumed by the device to extract bit values being processed. My research is aimed at evaluating the security of an Advanced Standard Encryption (AES) core which uses Wave Dynamic Differential Logic (WDDL) to defend against a Correlation Power Analysis (CPA) attack. To do this I used the ChipWhisperer Side Channel Attack platform to implement several of my AES designs onto an Field Programmable Gate Array (FPGA) and attacked them using a CPA attack. Since there are multiple stages of AES, I examined the security of various designs by implementing WDDL on individual stages, and also implementing WDDL on various combinations of stages. The results of my research were a 4538% increase in security of a fully implemented WDDL design on the AES core, but also an 805% increase in area of the design. These results exhibit the effectivenes of WDDL, but the additional area required highlights why this type of defense is challenging to implement on space limited systems.

I. Introduction

Embedded systems are application specific computers made to perform specific functionalities. They are responsible for a wide variety of applications, from automatic braking in a car to withdrawing money from an ATM. These systems are normally small devices with limited processing power, usually just enough to perform the functions required of it. Because of this type of design, embedded systems are often limited in the amount of security they can provide. They normally employ encryption algorithms but these encryption algorithms need to be as lightweight as possible, and sometimes this creates a security vulnerability.

An encryption algorithm is used to mask the data being processed inside the device. In certain applications of embedded systems, medical devices for example, the information being processed by the device may be sensitive and could be targeted by an outside attacker. Encryption algorithms, such as Advanced Encryption Standard (AES), have been shown to be resistant to cryptanalysis and therefore are mathematically secure for encrypting data. The problem when working with embedded systems is that an attacker can perform what is called a side channel attack to break these algorithms without using cyrptanalysis.

Side channel attacks target the hardware implementation of the encryption algorithm rather than the mathematical properties of the algorithm. These attacks target the electrical characteristics of the device and can infer certain properties of the device's current state and the data being processed. One type of side channel attack is power analysis. Different amounts of power is consumed depending on if the bit value being processed is a 1 or a 0. This means the total power being consumed by a device is related to the data being processed by it. Using this fact, attackers have the ability to capture numerous power traces, perform statistical analysis on them, and decipher specific bit values of the data being processed by the device. If this type of

attack is done when the device is running an encryption algorithm, the attacker has the ability to decipher bit values of the encryption key used to encrypt the data. If the attacker can extract all the bits of the key, then they obtain the ability to decrypt all the data being encrypted by the device.

My research was aimed at defending against these side channel attacks. More specifically, I focused on implementing the AES encryption algorithm in a way that is resistant to a side channel power analysis attack. My hypothesis was, if each gate in the design consumes a consistant power consumption for every evaluation, the device as a whole will have a consistant power consumption independent of the data being processed. To examine this hypothesis I worked with the idea of Wave Dynamic Differential Logic (WDDL). This type of defense adds a complement gate to every true gate in the design. With this type of design, when the true gate outputs a value, the complement gate will always output the opposite value. Another aspect of WDDL I implemented was a precharge phase. This precharge phase initializes the outputs of every gate in the design to 0. Therefore, when these gates are being evaluated, they will all be starting with the same initial value as their output. This will ensure that, between the true gate and its complement, there is only one transition of 0 to 1 regardless of the values being evaluated. If applied to every gate in the design, the result will be a consistent power consumption for the device and will in turn make the implementation resistant to power analysis [1].

My modified AES core was implemented on a Field Programmable Gate Array (FPGA). A FPGA is a programmable device used to mimic the functionality of an integrated circuit (IC). It implements programmable hardware, which is done by using lookup tables (LUTs). These tables take in specific inputs and depending on how they are programmed, produce specific

outputs. The ability for a designer to program the hardware on an FPGA makes it much more flexible than an IC and ideal for use in embedded systems.

To quantify the security of an AES core, I used the ChipWhisperer platform to conduct power analysis attacks on it. The ChipWhisperer platform allows a user to easily attack an encryption design programmed onto an FPGA. The ChipWhisperer software collects power traces and then analyzes them to extract the keys of the encryption algorithm [2]. The metric I used to compare the security of different AES core designs is the average number of power traces needed to uncover the encryption key. Also I wanted to examine the impact of area in these designs when implemented. To quantify this I recorded the number of LUTs in each design.

The results of my research showed the WDDL defense was effective in increasing the number of power traces needed to break the AES algorithm. An implementation of a fully WDDL protected AES core, when compared to the baseline AES with no added defense, had an 4538% increase in the number of power traces needed to uncover the encryption key. The downside of this was a 806% increase in area and a decrease in the clock rate by 50%. These results show the effectiveness of this design in terms of security but also highlights the issue of implementing this type of defense on space limited devices.

Section II gives background information about encryption algorithms, power analysis attacks, and several defences against theses attacks. Section III goes in depth on the research methods used throughout this year. Section IV displays the results of my research. The paper concludes with Section V.

II. Review of Literature

This section will start with a review of three commonly used encryption algorithms. It will then move into descriptions of three types of power analysis. Following that, several defences against power analysis will be discussed. It will conclude with a description of a defence implementation on an FPGA.

A. Standard Encryption Algorithms

There are three main encryption algorithms this paper is going to touch on, Data Encryption Standard (DES), Advanced Encryption Standard (AES), and Rivest-Shamir-Adleman (RSA). DES and AES are private key symmetric block ciphers and RSA is an asymmetric public key cipher. DES and RSA will be briefly mentioned, but the understanding of these algorithms will not be vital to the rest of the paper. The AES algorithm will be the focus of my research in following sections.

i. Data Encryption Standard

DES is an encryption standard block cipher that takes a 56-bit key and 64-bit plaintext block and outputs a 64-bit ciphertext block. DES is based on the idea of confusion and diffusion [3]. The SBox lookup tables are responsible for the confusion, which produce an output that has no visible relationship to the input. Diffusion is accomplished by multiple permutations and bitwise expansions to eliminate any repetitiveness that is present in the plaintext.

DES is a symmetric encryption algorithm. This means the same key is used to encrypt and decrypt data. The only modification in decryption is the round keys are issued in reverse order. Each block takes sixteen rounds to encrypt and in these rounds the plaintext is shifted,

permuted, expanded, condensed, and substituted to achieve an output that has no statistical correlation to the inputted data [4].

This algorithm was the first official standard cipher used in the commercial world, but as technology has advanced, brute force techniques have been proven to break this algorithm. The vulnerability in DES is that the algorithm only has a 56-bit key and, as the speed of computers has increased, it is now possible to run through all possible values of the key through simple guess and check [5]. A variation of DES has been developed called 3DES that uses the same algorithm but loops through it three times. This requires a 168 bit key, which is unbreakable through brute force techniques. The downside of using 3DES is that it is very computationally heavy. DES has to run three times for the data to be fully encrypted and this is often too much overhead for a device. This is where the motivation behind developing a new encryption standard, the Advanced Encryption Standard, comes from.

ii. Advanced Encryption Standard

AES, similar to DES, is a symmetric block cipher that was first standardized in 2002 [6]. AES operates on 128 bit blocks and offers an option of a 128, 192, or 256 bit key. There are four separate phases of AES, Key Expansion, Initial Round, Rounds, and Final Round. The following explanation assumes a key size of 128 bits, or 16 bytes, and will be looking at the encryption of a single block of 128 bits of plaintext. This process is close to identical to AES with 192-bit or 256-bit keys with some minor variations, including the addition of rounds and a modified Key Expansion phase. Figure 1 shows an overall representation of the AES rounds and dataflow between stages, which is discussed in [6].

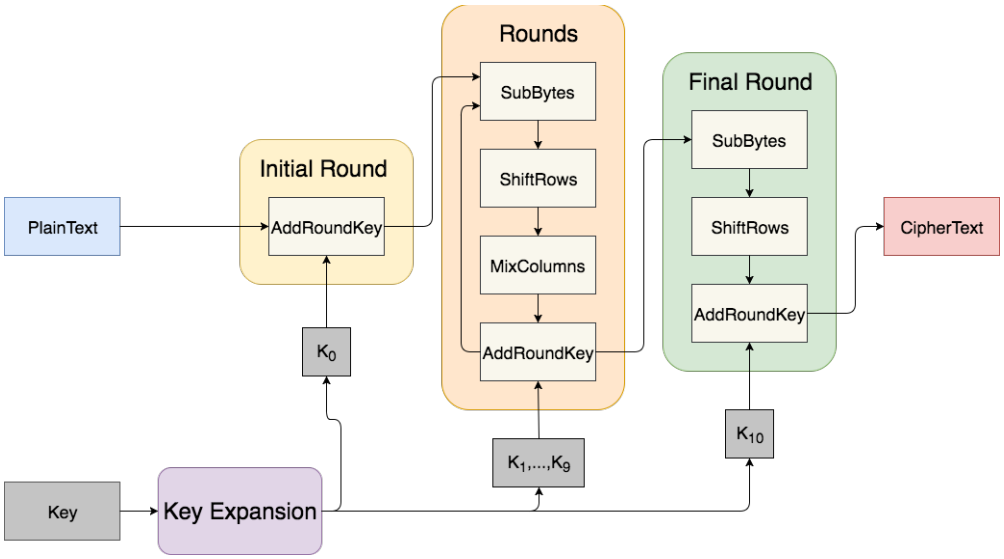


Figure 1: General overview of the AES encryption algorithm. The arrows represent the data flow through the algorithm. Each stage takes in 128-bit blocks and outputs 128-bit blocks. The Initial Round phase executes once, the Rounds phase executes nine times, and the Final Round phase executes once.

The first phase of AES is the Key Expansion. The role of Key Expansion is to map the private encryption key to multiple round keys, each 128 bits. There needs to be eleven round keys generated, nine for the Rounds phase, one for the Initial Round phase, and one for the Final Round phase. Key Expansion is done by first copying the initial 16 bytes of the original key into the first 16 bytes of the expanded key. Next, the least significant four bytes are rotated, substituted using a S-box, and then XORed with a value from the lookup table. The value from the lookup table is depended on the current iteration the algorithm is in. These four bytes are then XORed with the bytes 16 positions to the left of it and added to the end of the expanded key. This process is repeated until the entire expanded key is produced.

The next phase is the Initial Round phase. The only stage that occurs in this phase is AddRoundKey. In this stage the inputted plaintext is XORed with the a round key, K_0. The output of this phase is fed into the Rounds phase.

The Rounds phase has four different stages in it, SubBytes, ShiftRows, MixColumns, and AddRoundKey. The stages are repeated nine times before moving onto the Final Round phase. SubBytes consists of inputting the bytes into an S-box lookup table where they are substituted by their corresponding table values. Figure 2 shows a representation of the mappings of inputs of the S-Box to the outputs of the S-Box.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 2: Mapping of the byte substitution in the S-Box. The row is chosen by the four most significant bits and the column is chosen by the least significant bits. For example if 0x41 is inputted to the S-Box, the output will be 0x83 (after [6]).

The ShiftRows stage, shown in figure 3, does a simple shuffling of the bytes. By picturing the text being grouped as a 4X4 array of bytes, ShiftRows rotates all bytes right by the value of the row number minus 1. For example, the first row does not shift, the second row shifts right by one, and so on. The bytes shifted out of the array are wrapped around to the left side of the row.

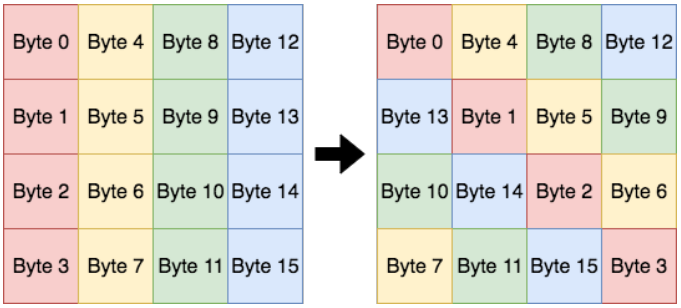


Figure 3: Visual diagram showing how the bytes are shifted in the ShiftRows stage. All bytes shifted out of the array to the right are wrapped around to the left of the array.

The output of the ShiftRows stage is fed into the next stage, MixColumns. MixColumns uses matrix multiplication to multiply the resulting 4X4 byte array by an array of predetermined constants.

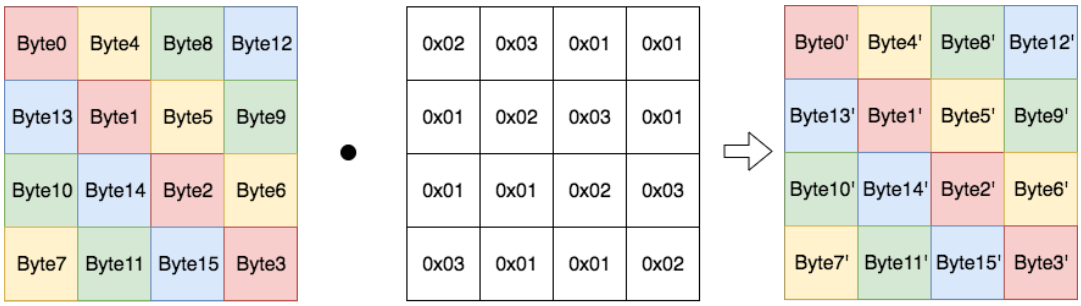


Figure 4: The MixColumn phase computes the dot product of the 16 bytes and an array of constants. The middle block represents the array of constants. These values are standard to the AES algorithm.

The last stage in this phase is the AddRoundKey, which is the same as described in the Initial Round phase. The only difference is the XORing is done with the round key associated with the specific round the algorithm is in.

Final Round is the final phase of the AES encryption algorithm. This phase consists of all the stages in the Round phase except the MixColumns stage. The output is the final ciphertext for the 128 bit block.

AES, although very effective in encrypting data, has some limitations. One of the major ones is the fact that it uses a private key for encryption and decryption. The use of a private key for encryption and decryption means that this private key needs to be securely delivered to each entity involved. Unless the private key can be physically delivered to each entity, the process of secure distribution becomes challenging. This is where the RSA encryption algorithm is used.

iii. Rivest-Shamir-Adleman

RSA is an asymmetric key encryption algorithm that uses a private key to encrypt data and a public key to decrypt data, or vise versa [3]. It is built on the foundation of prime number factorization and the challenges behind computing the factorization for large numbers. RSA is often used when information needs to be shared with an entity in which there is no private key shared.

B. Power Analysis

Although the algorithms described above have been shown to be secure against cryptanalysis, a new field of attacks has emerged that target the physical hardware of a system instead of the mathematics behind the encryption algorithm. These are referred to as side channel attacks.

Power analysis is a type of side channel attack that focuses on fluctuations in the power consumption of a device. From a power consumption trace, an attacker has the ability to observe specific deviations in power and, since this power is dependent on the data being processed by the device, the attacker can actually decipher bit values from a series of power traces. There are three main types of power analysis, Simple Power Analysis (SPA), Differential Power Analysis

(DPA), and Correlation Power Analysis (CPA). Each of these examine variations in power of a device and relate these variations to a system's state and information being processed by the system. The full understanding of SPA and DPA are not required to understand the rest of the paper. CPA was the main attack my research focused on.

i. Simple Power Analysis

Simple Power Analysis, discussed in reference [3], is the most basic of the power analysis techniques. SPA has two forms, single trace analysis and trace pair analysis. As hinted at by their titles, single trace analysis only uses one power trace and trace pair analysis uses two traces that are compared to one another.

In a device, different applications or tasks require different levels of power consumption. Single trace analysis examines one power trace at a time and picks out these variations in power. With some background knowledge on the device being analyzed, the attacker is able to decipher the state the system is in at specific times. On top of this, if there is minimal noise in the system, bit values can be distinguished using single trace analysis. For example, in some stages of RSA there is a choice of multiplication of values or squaring of values, and this is dependent on whether the data bit is a 1 or a 0. Multiplication uses significantly more power than squaring, therefore when there are spikes in power the attacker can assume the device is in a multiplication phase and can infer that the data bit being processed is a 1. On the contrary, when the power consumption is lower the attacker can assume the device is in a squaring phase and can infer the data bit is a 0.

In trace pair analysis, two power traces are examined together. This is usually done by subtracting one from the other. The result is a single trace that is relatively flat where the two

power traces were similar and sporadic where the traces are different. This data can be used to compare how different inputs affect the power consumption of the device.

SPA is mostly used in addition with other attacks. Many of these side channel attacks are aimed at deciphering a system's key to be able to decrypt the data being transferred. In a system that has moderate amounts of noise, reading individual bit values from a single power trace is nearly impossible. An example of when SPA is effective is when dealing with an unknown device, an attacker may use an SPA attack to learn what type of encryption algorithm is employed and then use a different attack to attempt to decipher the system keys.

Even though SPA is not used for deciphering keys, it is still a powerful attack. An example of a real-life SPA attack would be an attack on a security pin pad. A pin pad is a type of digital lock that requires the user to enter the correct code for it to be unlocked, and is represented by the state diagram in figure 5. If an attacker conducted an SPA attack on the device, given that each state will have a distinct power consumption, they could monitor how each of their inputted guesses affects the state of the device. The attacker will repeatedly make guesses on the correct code and observe the system's reaction to each guess. A change in power consumption will relate to a change in the state of the device. Through guess and check, and by knowing the power consumption of each state of the device, the attacker can execute multiple rounds of trial and error until the correct code is found.

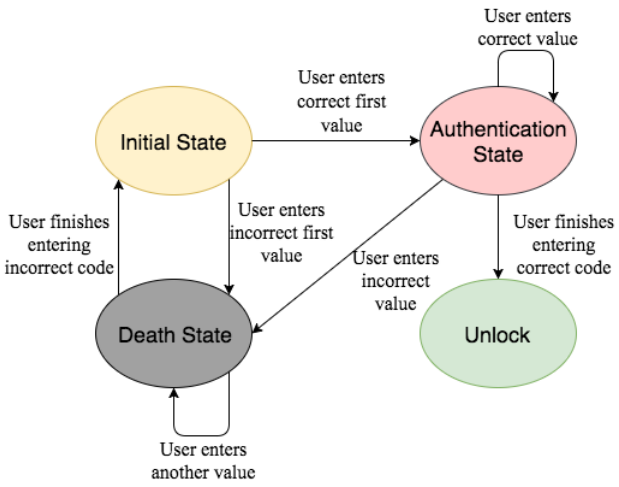


Figure 5: State machine of the pin pad being attacked. Because each state will have different power consumptions, these varying power consumptions can be monitored and used to attack the device.

Although SPA is a very effective attack to observe state changes in a system, it is not particularly effective in deciphering bit values. An easy defense to prevent SPA would be to add noise to the system. This would add minor fluctuations in the power trace that would make it challenging to pick out particular bit values.

ii. Differential Power Analysis

Differential Power Analysis is a type of power analysis that uses statistical methods to compare a very large set of power traces, pick out correlations between them, and use the correlations to decipher specific bit values. The idea behind DPA is to target a specific round of encryption, this round differs depending on the encryption algorithm used, and makes guesses to what the subkey value is. The following explanation, discussed in [3], will be aimed at an attack on the AES encryption algorithm.

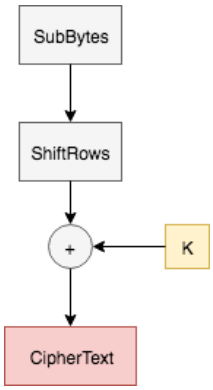


Figure 6: Final round of AES encryption algorithm. This is normally where where a DPA attack targets because the ciphertext is known and the only operations on the text are XORing and byte shifting.

A DPA attack usually targets the output of the SubBytes stage of the final round of the encryption algorithm, as shown above in figure 6. The reason the last round is targeted is because the MixColumn stage is not present in the final round of AES therefore all transformations are linear.

The main idea behind a DPA attack is to collect power traces, guess the 8-bits of the key value, and then using that, making a prediction on the output of the S-Box. Because the attacker is unable to physically see the actual output of the S-Box to confirm whether the prediction is correct or incorrect, the power traces need to be used to extract this data.

A DPA attack is executed as follows. First, the device is fed arbitrary inputs as plaintext and the plaintext is encrypted using the encryption algorithm. The ciphertext is recorded along with the power consumption trace of the device. Table 2 shows the resulting collection of data. In general, the noisier the system the greater the number of power traces that need to be collected.

Symbol	Description
j	Encryption run number
CT_i	Ciphertext of encryption run, j
Pow_i	Power trace of j
K	Key byte guess
I	Predicted S-Box output according to K
S	Selection function output (Most Significant Bit)

Table 1: Key mapping symbols to their description for the following DPA explanation.

j	CT_j	Pow_j
0	0010 0001	Pow_0
1	1000 1110	Pow_1
2	1011 0000	Pow_2
3	0010 1111	Pow_3
.	.	.
.	.	.
.	.	.
n	1010 0011	Pow_n

Table 2: For j encryption runs, power traces, Pow_j, are recorded along with their corresponding ciphertext, CT_j.

Once these power traces are collected, the attack moves into the key byte guess phase. Here, the attack uses the key byte guess, K, and the ciphertext, CT_j, to predict what the S-Box output, I, should be. Next, a selection function, S, is chosen. In this case the selection function takes in the predicted S-Box output and selects the most significant bit of the data. The power traces associated with the S-Box output guesses are used to check if the prediction was correct. All the power traces that, according to the key guess, should have a selection function output of 1 are placed in a set, Set 1. The same thing is done for the power traces that should have a selection function output of 0.

The power traces in each set are then averaged together. If the sets are split correctly, there is a specific point at which all power traces in Set 1 are computing a 1 value and all power traces in Set 0 are computing a 0 value. These two traces are then subtracted from one another. The variation in Set 1 and Set 0 will result in a spike after the subtraction. This spike indicates a correct key guess.

j	K	I	S
0 1 2 3 . . . n	0000 0000	0101 0111 1110 0011 1001 0101 0010 0000 . . . 1110 0101	0 1 1 0 . . . 1
0 1 2 3 . . . n	0000 0001	1000 0011 1010 1011 0000 1101 0111 0001 . . . 1100 1010	1 1 0 0 . . . 1
. . .			
0 1 2 3 . . . n	1111 1111	0111 0001 0011 1101 1001 1001 1010 1011 . . . 0001 0000	0 0 1 1 . . . 0

Table 3: Process of taking trace j and using key guess K to predict the output of the S-Box I. Selection function, S, in this case is simply selecting the most significant bit of I.

K	Set 0	Set 1
0000 0000	Pow_0, Pow_3, …	Pow_1, Pow_2, … , Pow_n
0000 0001	Pow_2, Pow_3, …	Pow_0, Pow_1, … , Pow_n
. . .		
1111 1111	Pow_0, Pow_1, … , Pow_n	Pow_2, Pow_3, …

Table 4: The power traces are separated into two sets. One for the traces that, according to the key guess K, should have a 0 as their selection function output and one for traces that should have a 1. The power

traces in each set are averaged together and then the set averages are subtracted from each other to examine correlation.

If the guess is incorrect, by the laws of probability, there will be close to 50 percent incorrect guesses in each of the sets. When the power traces in the sets are averaged, they will have generally identical results. Then when they are subtracted from each other, there will be no resulting spike. This will result in a flat line throughout the entire power trace representing an incorrect key guess.

Since the attacker is focusing on one byte of the key at a time, they will run through all the power traces collected 2^8 times for each key byte guess until the correct byte guess reveals itself. Once this is completed for one byte of the key, the attacker targets a different S-Box output and repeats the process until the entire last round subkey is uncovered. From there the attacker can work backwards through the key expansion algorithm to recreate the master key of the system. Once this master key is uncovered, the system is compromised.

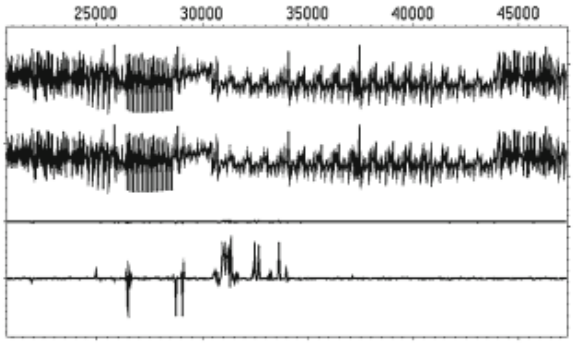


Figure 7: The top two power traces represent the average power consumption of Set 0 and Set 1. The third power trace shows the result of the subtraction of the top two traces. The fourth trace is a zoomed in version of the third. The spikes relate to differences in the average power traces of the sets and indicate a correct key guess (after [3]).

iii. Correlation Power Analysis

Another type of power analysis technique is Correlation Power Analysis (CPA). CPA is a version of DPA that focuses on all the values in a byte rather than just working with the output of a selection function [3][7]. The attack makes guesses on what a subkey byte would be, uses the known text to model what the power consumption would be if that key byte guess was correct, and then compares this model to the actual power trace. The subkey byte guess with the highest correlation is going to be the most likely key.

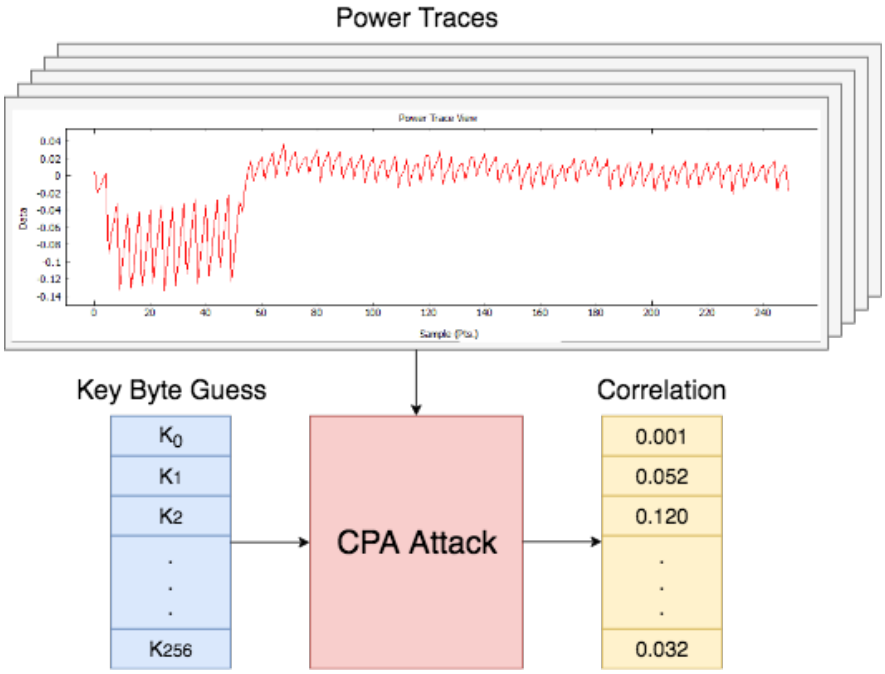


Figure 8: CPA attack takes in the power traces, recorded from the device while it runs encryption, and key byte guesses. The attack outputs correlations for each guess and the guesses with the highest correlation are the most likely key bytes.

This attack is made up of 4 different stages. The first is creating a model which allows for comparison between recorded power consumptions and generated power consumptions, obtained from using the key byte guess and the known text. In this case the most effective model is the

Hamming Weight model. Hamming Weight is the number of 1's existing in a bit stream. For example:

$$HammingWeight(00111001) = 4$$

A larger number of 1's being processed is going to correlate to a higher power consumption at that time. The Hamming Weight allows a comparison of data being processed and the power being consumed because of that data.

The next stage of the CPA attack is to record the power consumption traces of a device while it encrypts a number of random data streams. The power traces recorded will be compared against the modeled power consumptions of the key guesses. The noiser the system the more traces that will need to be recorded.

Once all the traces are recorded the next step is to use the traces to conduct the attack. This is done by examining one byte of the key at a time and taking guesses on what that key byte is. Since the attack only looks at a byte at a time, there are only 2^8=128 possible values of that byte. The attack iterates through each of these guesses and uses the known text to generate a model of what the power consumption would be if that key byte guess is correct. This is done for every possible value of that key byte. These models are then compared to the actual power trace using the equation below.

$$r_{i,j} = \frac{D\sum_{d=1}^{D} h_{d,i}t_{d,j} - \sum_{d=1}^{D} h_{d,i} \sum_{d=1}^{D} t_{d,j}}{\sqrt{\left(\left(\sum_{d=1}^{D} h_{d,i}\right)^2 - D\sum_{d=1}^{D} h_{d,i}^2\right)\left(\left(\sum_{d=1}^{D} t_{d,j}\right)^2 - D\sum_{d=1}^{D} t_{d,j}^2\right)}}$$

- t → power trace
- D → total power traces
- d → individual trace
- j → sample point in trace
- i → key byte guess
- h → power estimate
    - h(d,i) power estimate for trace d and guess i

Using this equation the attack finds the highest value of $| r(i,j) |$. From there, it finds the highest $| r(i) |$ and the key byte guess results in this value is going to be the most likely key.

The final step of the attack is to repeat this process for each byte of the key. At the end, the collection of key byte guesses is the most likely round key and from there the attacker can work backwards to find the overall encryption key.

C. Defence Implementations

These power analysis attacks have the ability to compromise a system without performing any cryptanalysis on the encryption algorithm used. Systems that run these encryption algorithms need to examine not only the encryption algorithm used but also the implementation of the algorithm on the device. The following subsections discuss different implementations of security measures to defend against these side channel power analysis techniques.

i. Tamper Resistant Systems

One way to defend against an attack on a system is by being able to sense when an attack is happening. The design proposed in [8] uses a Security Primitive Controller (SPC) to implement the systems security operations. Their design also includes a System Security Controller (SSC) whose main purpose is to detect when the system is being attacked. This controller uses sensors connected to the battery, the bus, other communication channels, and other security primitives to detect out of the ordinary behaviors resulting from fault injections or abnormal operations.

The SSC has the ability to interrupt the SPC. Depending on the irregularity the SSC senses in the system, it will instruct the SPC to perform a specific function. This type of system defends against power analysis by detecting when the power consumption of the device is being monitored by an outside source. The SSC will detect this intrusion and will instruct the SPC to halt the encryption module. By halting the encryption module, the power traces collected by the outside entity will not be sufficient for data analysis.

Being able to alert a system when it is being attacked is a very viable defense against attacks, but it comes at a cost. Implementing this extra SSC controller and the sensors that go along with it will increase the hardware and complexity of the system. Along with this, having the SSC constantly polling these sensors to detect irregularities will use extensive amounts of power.

ii. Randomness

All defenses implemented to protect a system from an outside observer are aimed at masking the processes being executed in the system. Along with this, all side channel attacks can be executed because of consistent patterns in data. One way to combat these attacks is by adding an element of randomness to the timing of the system. An element of randomness adds unpredictability that cannot be modeled by an attacker. This approach is discussed in [3].

Attackers need to observe a system for a period of time before being able to successfully attack it. They look for patterns based on time intervals and correlate these patterns to specific functions in the system. By randomizing the amount of clock cycles before an instruction executes, there will be no clear time interval for each operation. Irrelevant instructions can be inserted into the instruction path to delay the execution of the thread. An attacker will not be able

to tell which instructions are placeholders and which are relevant to the thread's execution. This defence works especially well against power analysis. The varying number of clock cycles needed for execution throws off the alignment of power traces making them significantly more challenging to compare to one another.

Randomizing clock cycles is an easy design to implement in a system and is very low cost in terms of hardware. The downside is that it hurts the performance of the processor. When the processor is executing all these pointless instructions, it is not processing the meaningful ones. These relevant instructions get delayed to a later time and the system does not run as efficiently.

iii. Dynamic and Differential Logic

Another defence against a power analysis attack is Dynamic and Differential Logic (DDL). DDL allows the evaluation of each gate to have exactly one transition. This occurrence of a single bit flip is independent of the data being processed by the gate. By only having one bit flip in each gate, regardless of the data being evaluated, that cell will always consume approximately the same amount of power. To do this, DDL uses two main ideas. One, for each gate being evaluated, also evaluate its complement. The true gate and its complement together make a DDL cell. The second main idea is, the outputs of each gate are precharged to a set value before evaluation takes place. The precharge phase is used to initialize every gate output to a constant initial value. Because each gate output is set to the same value, when a DDL cell is evaluated, either the true gate or its complement will have a bit flip, but never both.

Sensor Amplifier Based Logic (SABL) is a type of Dynamic and Differential Logic and is used on Application Specific Integrated Circuits (ASIC) [1]. In SABL, there are two phases, a

precharge phase and an evaluation phase. The output of each individual gate is ANDed with the precharge value. Reference [1] explains, “whenever the precharge signal is 1, the outputs are predischarged to 0 independently of the input-values.” The problem with this approach is the design area is essentially quadrupling. For every gate in the design, one, its complement gate needs to be added and two, an AND gate has to be added for every true gate and for every complement gate.

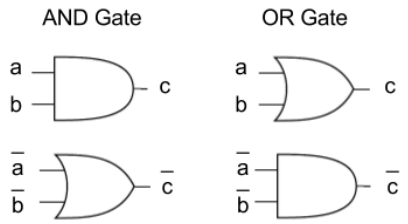


Figure 9: Representation of an AND gate and an OR gate in WDDL format. Both the true value and its complement are evaluated.

Another approach to DDL is Wave Dynamic Differential Logic (WDDL). WDDL is an implementation of DDL in which the precharge phase is initiated by forcing all inputs to 0. By doing this, each gate will evaluate to 0 and will have a precharge value of 0 at their outputs. This 0 will propagate through the entire design, like a wave [9]. WDDL uses the same idea of having a complement gate for each true gate, making a WDDL cell, and therefore, will still only have one bit flip for every evaluation of that cell.

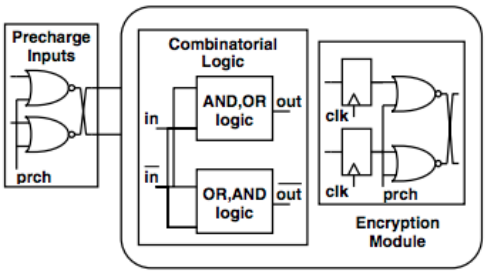


Figure 10: The precharge signal is applied to the NOR gates along with the inputs to the encryption module. If the precharge signal is 1, the input to the encryption module will always be 0. This triggers the wave which propagates throughout the system (after [1]).

The advantages of using WDDL are one, it can be implemented on an FPGA, and two, it does not require the addition of an AND gate for every gate in the design. The precharge phase is instead initiated by adding *n* NOR gates, where *n* is the number of inputs to the design. Each NOR gate takes in an input to the design and the precharge signal. Figure 10 shows a representation of this set up. Whenever the precharge signal goes high, the output of the NOR gate will always be 0 and this will be propagated into the design. This forces all gate outputs in the design to 0 [1]. By precharging before evaluation, each cell will only have one transition of 0 to 1 during the evaluation phase and will in turn have a constant power consumption that is independent of the data.

D. FPGA WDDL Implementation

A WDDL design can be implemented on an FPGA as described in [9]. There are a couple challenges when implementing such a design on an FPGA that make it more difficult than an implementation on an ASIC. One of the challenges is, when programming all these gates on the FPGA, the FPGA will try to optimize the design. It will look to combine these gates into the same lookup tables (LUT) to reduce the area of the design. This is an issue when trying to

implement a WDDL design because in WDDL the main goal is to balance load capacitances of each cell along with the routing between cells. Optimizing the design by combining gates into the same LUT will result in unbalanced load capacitances, which will in turn result in fluctuation in power consumptions related to the data being processed.

In WDDL there are two aspects of load capacitance that need to be considered. The first is the load capacitance of the LUT and the second is the load capacitance of the routing from LUT to LUT. Fortunately, two LUTs with the same number of inputs and outputs have identical load capacitances. The aspect that the designer needs to worry about is the load capacitance of the wires routing the output of LUTs to different LUTs. An approach for balancing this capacitance between a gate and its complement gate is by instantiating each gate to its own individual LUT and then combining these two LUTs into a single slice in the FPGA to represent a WDDL cell. A slice is a section of an FPGA consisting of a number of LUTs, multiplexers, and registers.

This is an effective approach because as stated before, the LUTs will have identical load capacitances. Along with this, one of the properties of a slice is that the routing from one LUT to the output of the slice is the same distance as the routing of the other LUT to the output of the slice. Since they are the same routing distance they will have the same load capacitance within the cell. A constant load capacitance within the WDDL cell will result in a constant power consumption by the cell.

Another challenge of a WDDL design on an FPGA is the amount of area it takes up. If a designer were to implement a WDDL design such as the one described above, one, the instantiation of each gate to an individual LUT would greatly increase the area the design takes up, and two, by also including each gates complement, the area would double on top of that. This

becomes a serious concern when the FPGA is suppose to handle other applications along with encryption.

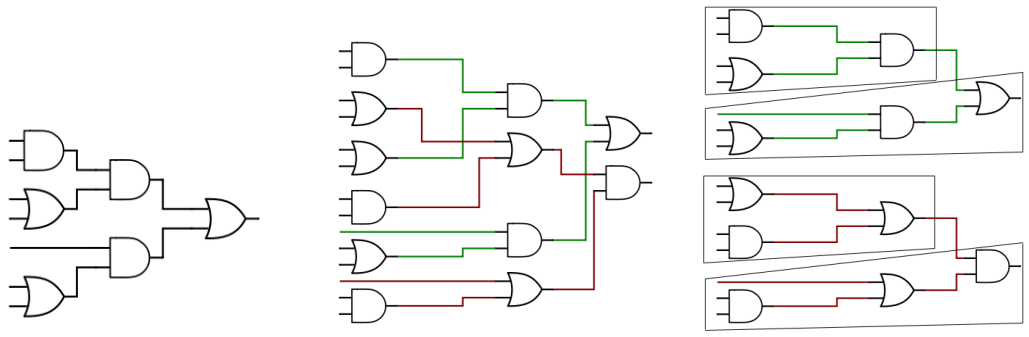


Figure 11: By implementing WDDL on the initial circuit shown in the first figure, result shown in the second figure, multiple gates can be grouped together to potentially be instantiated to the same LUT. This grouping reduces the area of the design.

One way to work towards reducing this amount of area in a WDDL design is by combining multiple gates into a single LUT. The figure above shows a small section of a design with three OR gates and three AND gates, along with its WDDL implementation. As seen in the figure, the gates generating the true values can be separated from the ones generating their compliments. By doing this, a designer can implement these two different components in four LUTs with four inputs and one output each. Now, instead of having twelve LUTs in the design, there are only four.

III. Methods

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 the Correlation Power Attacks (CPA) attack 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 AES 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 was the CW1173 ChipWhisperer-Lite and the

target board I used was 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 varied. The baseline design was a gate-level AES core with no additional defences added to it. From there I implemented just the logic and LUT instantiation aspect of WDDL, without precharge phase, on the different stages along with various combinations of the stages. After this I implemented the logic and precharge phase on the entire design and then compared the security variations different stages to LUTs. All of these implementations were attacked using the ChipWhisperer platform and the resulting number of power traces needed to uncover the keys were recorded.

Figure 12 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.

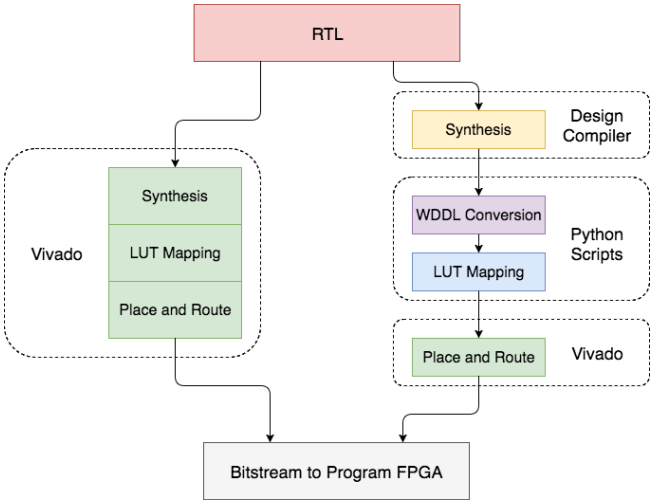


Figure 12: Overview of research components starting with RTL design all the way to the bitstream that programs the FPGA. The left side allows Vivado to perform all the LUT mapping and Place and Route. The right side, my approach, handles the mapping of gates to LUTs to avoid any optimization that may be done by Vivado. This ensures a balanced load capacitance 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 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 generate a bit stream to be programed onto 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 these traces to conduct attack.

The capturing of power traces from the ChipWhisperer CW305 target board is done by completing 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 13 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.

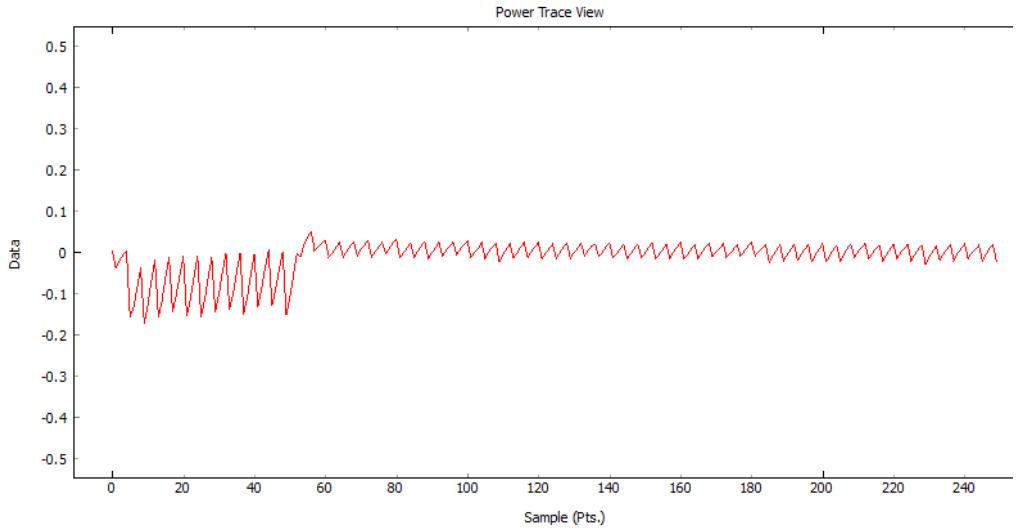


Figure 13: Power trace of one run of the AES algorithm. The power spikes that start the power trace represent the eleven rounds of the AES algorithm.

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.

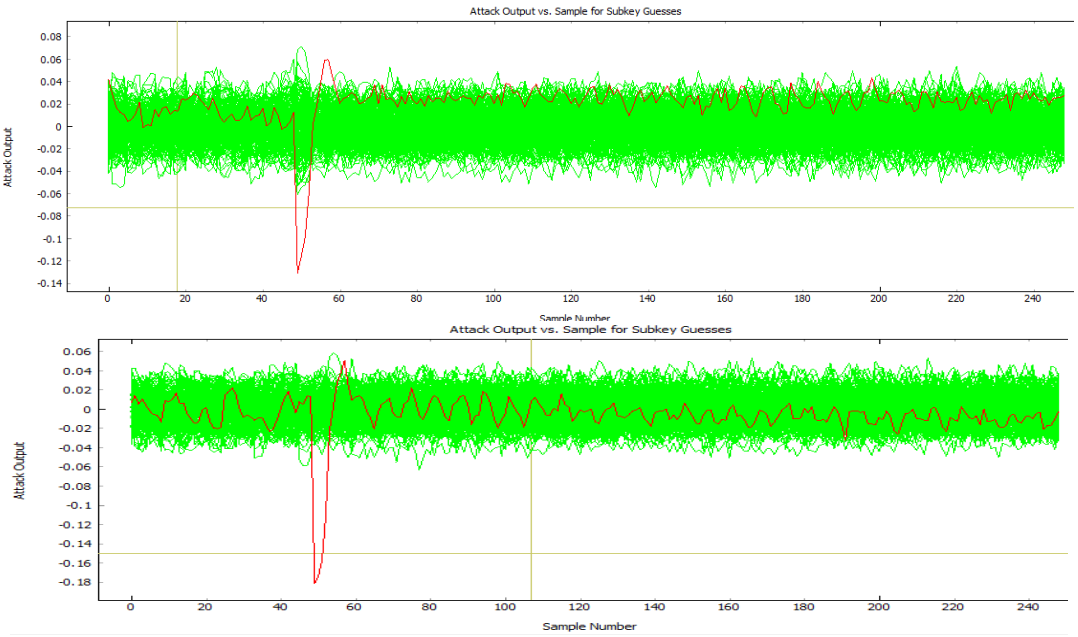


Figure 14: The spike around sample number 50 represents a correct key guess for both byte 0, shown in top figure, and byte 7, shown in bottom figure. The spikes occur at the same point in each plot because this is the point in the power trace where the output of the S-Box is evaluated and where the attack 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 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.

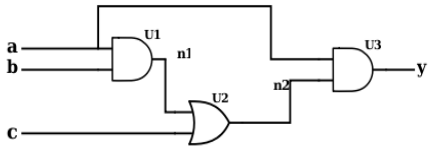
Once I had the gate-level AES design, the next step was to write a Python script to parse this file. The main goal of this script was to be able to convert specified modules in the AES core to WDDL format. The user inputs the names of the modules that need to be converted and the script parses the AES 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 protected by WDDL. If the module was to be protected using WDDL 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.



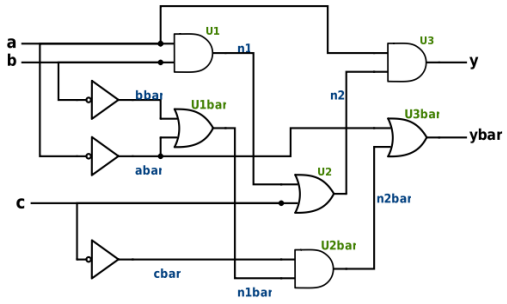

```
module sample_circuit ( a, b, c, y );
  input a, b, c;
  //input_done

  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 15: 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 16: 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 were 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 protected. 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 17: 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 macro 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 output how many power traces the attack needed to successfully uncover the keys. The ChipWhisperer Analyzer program takes in all the power traces recorded, iterates through all of them during the attack, and the results are updated in the results table, shown in figure 18, as the attack goes 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, I would have had to watch the results table throughout the entire attack. This was not possible for an attack that may last several hours.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	0	105	46	58	150	9	25	1	1	5	27	0	0	0	0	0
0	D0 0.2179	33 0.2349	84 0.2389	25 0.2170	F9 0.2164	21 0.2189	B9 0.2025	85 0.2240	2F 0.2360	2F 0.2189	EE 0.2164	C8 0.2660	B6 0.2350	63 0.2242	0C 0.2330	A6 0.2267
1	CC 0.2016	22 0.2016	D1 0.2007	53 0.1969	0A 0.2104	DC 0.2129	9A 0.2005	89 0.2222	E1 0.2112	C5 0.2050	BB 0.2011	91 0.2251	09 0.2185	EA 0.2083	4D 0.2003	D2 0.2089
2	8F 0.2012	7D 0.1926	15 0.2004	94 0.1934	6E 0.2014	2A 0.2021	A3 0.1984	45 0.2162	CE 0.2101	CE 0.1932	E2 0.1988	CC 0.2071	55 0.2147	8A 0.2037	B8 0.1981	B6 0.2019
3	B6 0.2003	9F 0.1898	4D 0.1994	D1 0.1922	6C 0.2000	03 0.1926	92 0.1949	99 0.2160	38 0.2089	86 0.1924	C0 0.1985	34 0.1991	32 0.2063	9B 0.1958	3B 0.1950	A1 0.1929
4	40 0.1994	D9 0.1862	63 0.1967	0F 0.1861	B9 0.1985	A6 0.1926	65 0.1906	F1 0.1998	FB 0.2035	4D 0.1906	D1 0.1954	8E 0.1984	0D 0.2042	3E 0.1950	7F 0.1925	7D 0.1925
5	53 0.1985	F1 0.1857	7F 0.1893	52 0.1817	78 0.1876	58 0.1909	58 0.1879	EF 0.1906	64 0.1951	3F 0.1896	A5 0.1906	51 0.1946	C9 0.1972	37 0.1949	87 0.1864	42 0.1864

Figure 18: 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 figures 19 and 20. 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 checked 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 correct subkey guesses were in the top spots. Once all 16 subkeys were in the top spots I set a threshold stating, if these correct subkeys all stay in the 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.

```
#########################################################################
if(subKeysFound == 16):
    self.file.write("\n\n\n############################\n")
    self.file.write( "found all subKeys\n" )
    self.file.write("Iteration: " + str(self.iterations) + "\n")
    self.file.write("\############################\n\n\n")
    self.subKeyIter = self.subKeyIter + 1
else:
    self.subKeyIter = 0

if( self.subKeyIter == 20 and self.done == False ):
    self.file.write("\n\n\n############################\n")
    self.file.write("DONE\n")
    self.file.write("Traces: " + str(self.iterations*10))
    self.file.write("\n############################\n\n\n")
    self.done = True

self.setVisible(True)
#########################################################################
```

Figure 19: 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.

```
#new
#gives value for each subkey per iteration, index1 is the xaxes, index2 is yaxes
self.file.write( str(maxes[j]['hyp']) + " " + str(highlights[index1]))
if( maxes[j]['hyp'] == highlights[index1] ):
    if( index2 == 0 ):
        subKeysFound = subKeysFound+1
        self.file.write( "\n\n********************\n")
        line0spot = "First Spot\tindex1: " + str(index1) + "  index2: " + str(index2) +
                    "    " + str("%02X\t%.4f" % (maxes[j]['hyp'], maxes[j]['value']) + "\n")
        line1spot = "Second Spot\tindex1: " + str(index1) + "  index2: " + str(index2+1) +
                    "    " + str("%02X\t%.4f" % (maxes[j+1]['hyp'], maxes[j+1]['value']) + "\n")
        self.file.write(line0spot)
        self.file.write(line1spot)
        self.file.write(str(subKeysFound))
        self.file.write("\n********************\n\n")
#########################################################################
```

Figure 20: 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 attacked 5 times and the average number of power traces to uncover the key was 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 attacked in the tutorial. This substitution was relatively simple and the only modifications that needed to be made to the gate-level 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. I developed a wrapper for the gate-level AES core that handled these signals but when the design was programmed on the FPGA and attacked, the following power trace was recorded.

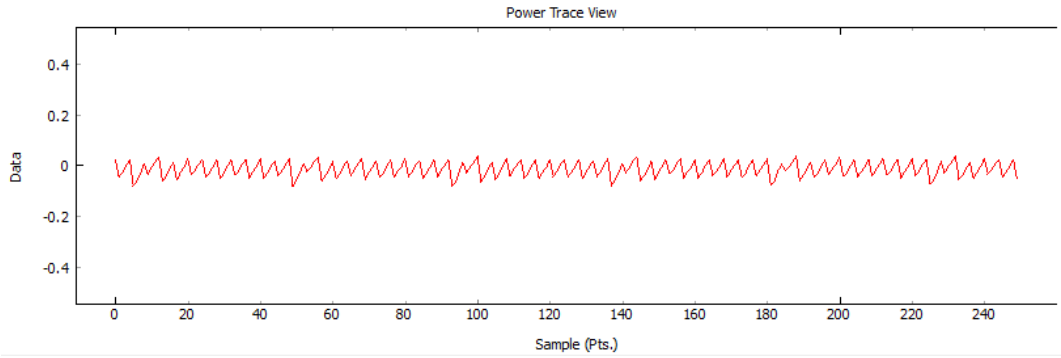


Figure 21: Single power trace of implemented AES core. The eleven peaks repeat five times representing the encryption algorithm running five times.

I collected 15,000 power traces and after the attack iterated through all of them the correct key was not uncovered. Although this may seem like a good sign for the design, considering this AES core was essentially a gate level version of the tutorial design, the attack

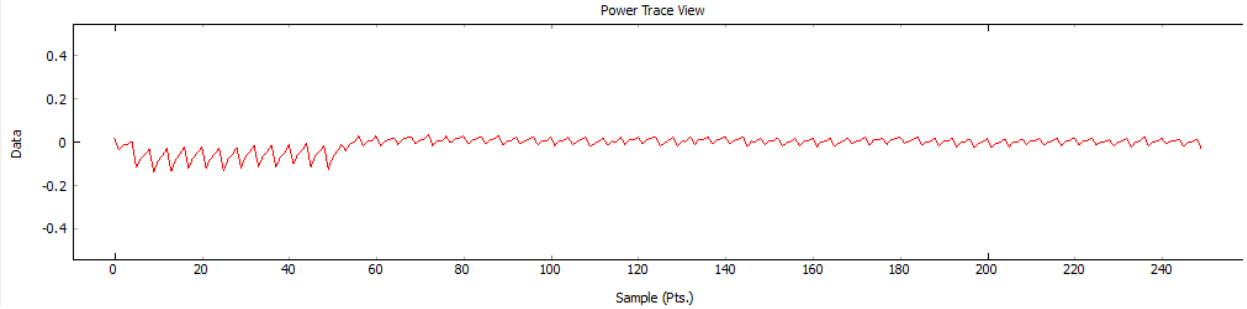
should have completed relatively quickly and the correct key should have taken no more than 5,000 power traces to uncover. This told me that the power traces being compared were not lining up correctly meaning the signals I incorporated into the design were not being triggered at the right times.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	62	229	133	181	219	208	138	213	100	181	142	112	90	147	48	211
0	FA 0.0097	AD 0.0113	47 0.0101	A6 0.0099	7C 0.0128	7F 0.0108	22 0.0095	5D 0.0103	6A 0.0107	9B 0.0101	CE 0.0104	8A 0.0116	12 0.0112	4A 0.0112	93 0.0108	B0 0.0105

Figure 22: 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 one I incorporated the gate-level design into the default design. This was done by deleting the sections in the default AES design that were related to the SubBytes stage, the MixColumn stage, and the AddRoundKey stage. I replaced these sections with the gate-level AES stages from the gate-level AES design. This ensured the AES core was receiving the appropriate signals for the ChipWhisperer software to start the encryption at the same time for every trace resulting in a design that the ChipWhisperer Analyzer program could attack. This ensures the traces line up allowing the Analyzer software to attack the design. Along with that, it also ensured the gate-level stages could be modified and protected with WDDL. I was planning on modifying the ShiftRows section too, but after further examination, modifying this section would not make a difference since it contains no AND or OR gates.

After substituting sections of the gate-level design into the default design I was able to successfully attack it. Figure 23 shows a single power trace from the encryption. This trace 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. 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 the 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.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	D0 0.1037	14 0.1724	F9 0.1604	A8 0.1821	C9 0.1238	EE 0.1806	25 0.1496	89 0.1813	E1 0.1178	3F 0.1764	0C 0.1465	C8 0.1357	B6 0.1372	63 0.1613	0C 0.1451	A6 0.1549
1	8C 0.0536	43 0.0569	3A 0.0523	3F 0.0682	4D 0.0565	84 0.0558	DB 0.0557	7F 0.0533	77 0.0566	E7 0.0557	E9 0.0601	96 0.0621	E8 0.0554	C2 0.0594	43 0.0544	E6 0.0613
2	FD 0.0523	54 0.0535	4C 0.0518	7C 0.0621	7F 0.0541	D5 0.0548	BC 0.0544	CA 0.0520	B7 0.0556	56 0.0554	24 0.0585	EB 0.0600	25 0.0536	45 0.0584	FC 0.0527	D6 0.0567

Figure 23: 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. The average correlations of the correct subkey guesses is 0.141.

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.

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. This module is 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 was the NORing of the inputs with the clock signal, this is how the precharge phase is implemented. This NORing 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.

An example of how a simple circuit is changed to a fully protected WDDL is shown in figure 24. The clock signal is now included in the module design as well as NOR gates. 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.

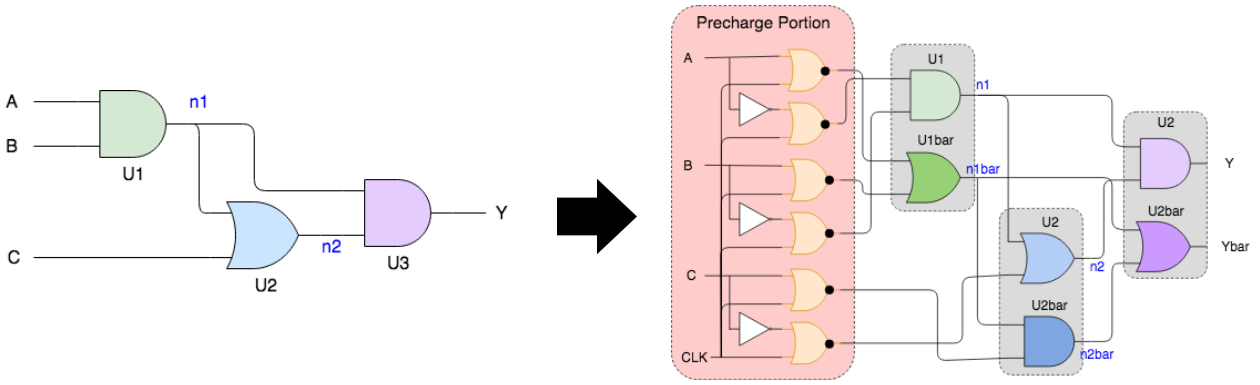


Figure 24: A fully protected circuit has gate complements and precharge phase. The clock and NOR gates are needed to implement the precharge phase.

IV. Results

A. WDDL With No Precharge

The table below shows the results of the attacks for a WDDL implementation with no precharge phase. The following table shows the AES design being attacked along with the area and average traces. The stages used for the name indicate the stages which have WDDL implementation and each gate is instantiated to its own LUT.

AES Design	Area	Traces Avg.	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Basic	2055	1526	1250	1430	1880	1640	1430
AddRoundKey	2224	2338	3060	1840	2550	2320	1920
MixCol	3510	2386	2040	2550	2030	2240	3070
SubBytes	8936	3374	2790	3940	3170	3000	3970
Add+Mix	3654	3000	2380	2720	3130	3290	3480
Add+SB	9780	3452	3360	4580	2520	3680	3120
Mix+SB	10487	6600	7540	5310	5130	8450	6570
All	11077	4450	5340	3190	4290	5330	4100

Table 5: Results of WDDL modification without precharge implementation.

One interesting aspect to observe in the data above is that there is a general trend where the security of a design increases as the area increases. Figure 26 displays the data in a scatter plot. From this graph one can see this general trend. There are two explainations for this. One, the added logic of WDDL is going to add system noise. The more noise in the system, the more traces needed to successfully conduct a CPA attack on the design. The other explaination is that by adding the complementary logic of WDDL in the different stages of AES, the balance of 0's and 1's being processed in the design begins to equalize. This equalization is going to translate to less variation in the power consumed by the device resulting in more security against a power analysis attack.

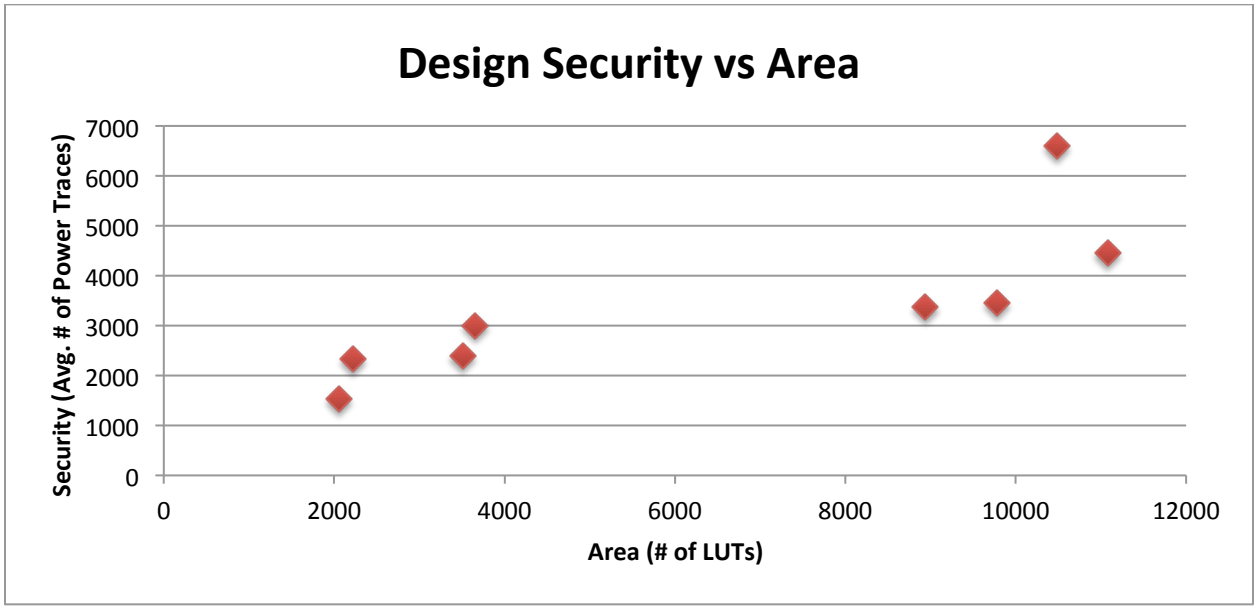
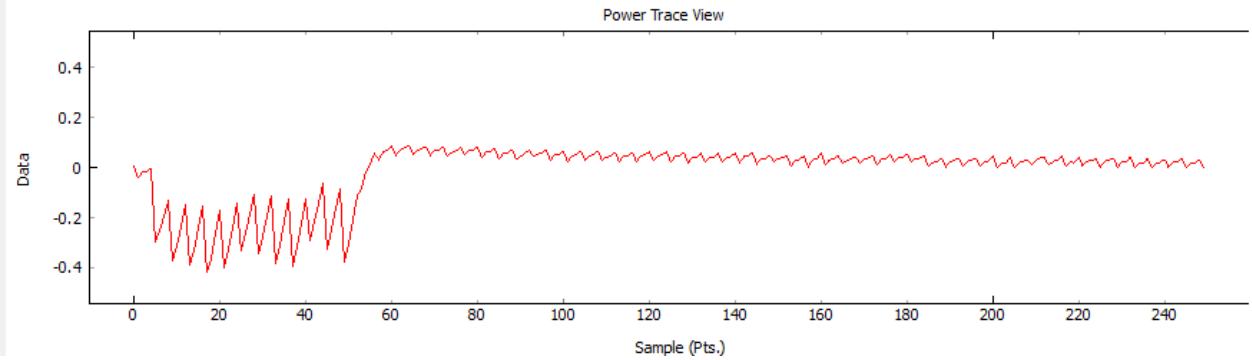


Figure 25: Scatter plot shows the interesting trend of an increase in area correlating with an increase in security.

The power trace for the design with all stages of AES implementing the logic of WDDL is below. This trace still looks very similar to the trace of the baseline attack but the correlations of the subkey guesses are lower. The average correlation for the baseline AES design was 0.141,

the average correlation for the AES with WDDL implemented on every stage was 0.104. The lower correlation of the subkey guesses demonstrates less variation in the power traces.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	D0 0.0887	14 0.1200	F9 0.1150	A8 0.0967	C9 0.0647	EE 0.1002	25 0.1348	89 0.1445	E1 0.1006	3F 0.1349	0C 0.0734	C8 0.1179	B6 0.1123	63 0.0849	0C 0.0977	A6 0.0914
1	CD 0.0665	62 0.0562	F0 0.0503	7C 0.0523	FE 0.0602	8B 0.0564	8E 0.0572	58 0.0550	4D 0.0582	BC 0.0520	8D 0.0520	E1 0.0510	20 0.0642	6E 0.0482	AA 0.0527	F5 0.0493
2	3E 0.0541	6D 0.0521	A7 0.0490	A1 0.0496	BA 0.0546	00 0.0527	C6 0.0549	9D 0.0528	ED 0.0576	C8 0.0513	58 0.0511	75 0.0490	59 0.0637	9D 0.0482	24 0.0507	F6 0.0492

Figure 26: Power trace of design with fully implemented WDDL logic with no precharge phase. The average correlations of the correct subkey guesses is 0.104. Attack completed in 4450 traces.

B. WDDL With Precharge

The table below shows the results of the attacks for a WDDL implementation with no precharge phase. The AES designs in the table are labeled by the stage of AES which gates are instantiated to their own LUTs. The WDDL logic and precharge phase are implemented through out the entire design.

AES Design	Area	Traces Avg.	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Basic	11912	1060	910	1150	1270	1000	970
AddRoundKey	12280	1010	780	880	1080	1130	1180
MixCol	12932	988	1020	1000	1020	1030	920
SubBytes	17222	38313	38860	52300	34310	28110	37985
Add+Mix	13128	1020	1020	980	1000	1110	990

Add+SB	16664	40658	43900	38790	38400	40130	42070
Mix+SB	18324	53050	44600	62100	55030	50430	53090
All	18618	70440	84230	69700	58400	65280	74590

Table 6: Results of WDDL modification without precharge implementation.

The results seem to vary in this table depending on what stage of the algorithm is protected by WDDL. Each of the designs which have WDDL implemented on the SubBytes stage have a significant increase in security. This is visible when only SubBytes is protected along with when it is protected in combination with AddRoundKey or MixColumn. In the design where only AddRoundKey was protected, it only took an average of 1010 power traces to uncover the key. When AddRoundKey and SubBytes are protected together, this number jumps up to 40,658. This is an increase of 40x.

One explanation for why designs that have a protected SubBytes stage have a significant increase in security is because this stage is the most computationally heavy. When WDDL is implemented in the SubBytes stage there are on average an addition of 5300 LUTs. This is compared to MixColumn, which only has an addition of about 1000 LUTs, and AddRoundKey, which only has an addition of about 400 LUTs. This means, by protecting the SubBytes stage, WDDL is actually protecting the majority of the essential logic in the AES core and therefore providing the most security.

Along with this explaination I also have a hypothesis for why the increase in security is so dramatic. The steep increase in security when protecting the SubBytes stage is realted to the fact that this stage appears in the last round of AES which also happens to be the point of attack for this specific attack. This hypothesis is a little bit shaky because AddRoundKey is also in the last round but protecting this stage does not seem to have an impact on the security of the

algorithm. I suspect this is related to the fact that the AddRoundKey stage has very little logic in it and therefore protecting this logic causes minimal affect on the overall security of the algorithm.

The other interesting aspect of the data that was collected is the basic design with a precharge phase actually has about half the security of the baseline design with no added defenses. This is also consistent with any design that does not protect the SubBytes stage. I was not able to explain why was happening. If I were to continue this research, this would be the first aspect I look into because if protecting only the SubBytes stage makes a difference in security, the area of the final design could be reduced without sacrificing much security by only protecting SubBytes.

The graph below shows the optimal designs for WDDL protection. Any of the designs below the line exhibit an increase in area without an increase in security. A design that meets this condition is not an optimal AES design.

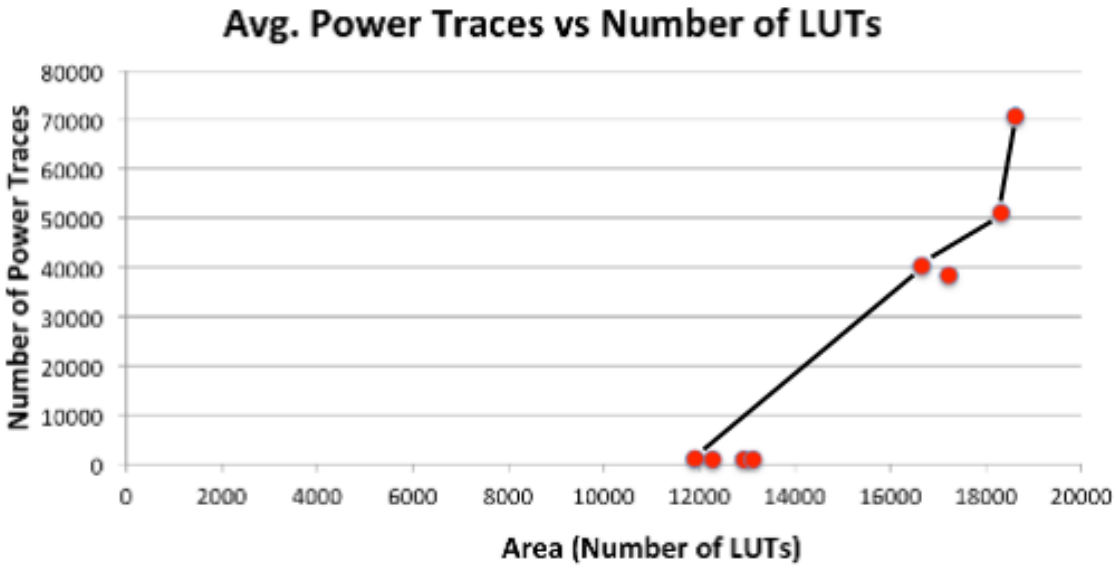
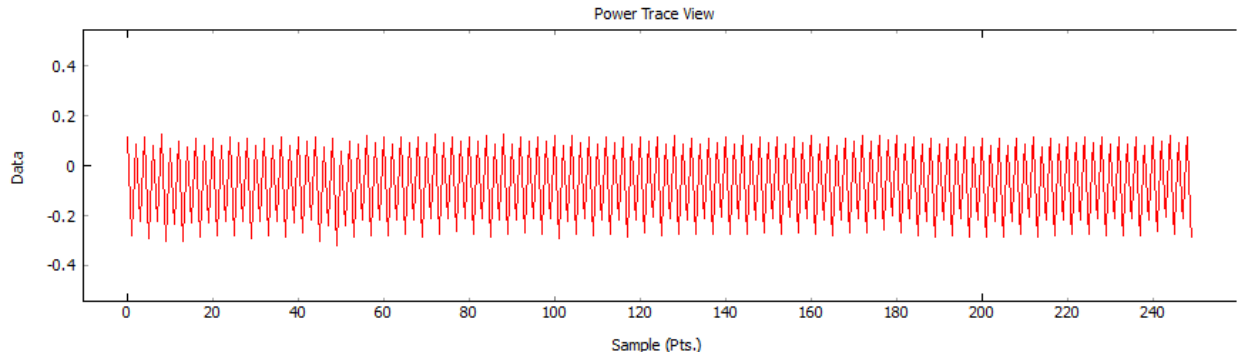


Figure 27: Graph displays optimal designs when protecting AES with WDDL.

A fully WDDL protected AES core took on average 70,440 power traces to uncover the subkey. This attack took about 4 hours to complete, which is a significant increase compared to the basic AES core which only took 20 minutes to break, and the resulting power trace is shown in figure 28. Along with this, the design has 46 times the security as the baseline AES core. With this increase in security comes an increase in area as well. When compared to the baseline AES core, the fully protected AES core has 9 times the area.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	D0 0.0140	14 0.0105	F9 0.0113	A8 0.0110	C9 0.0103	EE 0.0123	25 0.0078	89 0.0160	E1 0.0109	3F 0.0136	0C 0.0091	C8 0.0135	B6 0.0101	63 0.0146	0C 0.0117	A6 0.0063
1	E1 0.0073	E0 0.0049	CF 0.0059	9D 0.0047	4C 0.0098	30 0.0048	B0 0.0056	01 0.0058	4C 0.0080	F4 0.0049	31 0.0055	2E 0.0058	4C 0.0089	AD 0.0051	CF 0.0051	38 0.0051
2	8F 0.0072	F4 0.0047	6D 0.0048	3D 0.0047	95 0.0086	8B 0.0046	A7 0.0047	F7 0.0049	46 0.0073	C2 0.0046	2C 0.0052	CE 0.0051	CA 0.0075	CB 0.0047	40 0.0047	96 0.0049

Figure 28: Power trace for fully protected AES core shows a more consistant power consumption. The additional spikes, not seen in previous design traces, are related to the device switching from precharge to evaluation phase over and over again. The results table show a successful attack but the correlations exhibit the increase difficutly of attacking this design compared to others. The average correlation of the correct subkey guesses is 0.0108 which is a over 13 times smaller than the average correlation of baseline AES design guesses.

V. Conclusion

This year my research was aimed at improving the security of an AES-128 design implemented on a FPGA when defending against a CPA attack. To do this my approach was to evaluate the protection WDDL provides to an AES core when defending against a power

analysis attack. Data was collected for protecting different stages of the algorithm, along with incorperating different aspects of WDDL.

A fully protected AES core using WDDL with a precharge phase took on average 70,440 power traces to uncover the encryption key. This is a 46x increase in security when compared to the basic AES core which took 1526 power traces to break. The down side of implementing this type of defense is a steep increase in area. The fully protected AES core used 18,618 LUTs and when compared to the basic AES design, which had 2055 LUTs, this is a 9x increase in area.

This increase in area highlights one of the main issues with security in embedded systems. Although there are some defenses that can be implemented to defend against side channel attacks, the expense of area is often too much for the device. As stated earlier, these embedded systems are usually application specific and made as small and efficient as possible. This means any increase in size, like the one witnessed in the data, would mean an increase in size of the device which is not always an option.

References

[1] Tiri, K., Verbauwhede, I., Proceedings. Design,Automation and Test in Europe Conference and Exhibition, "A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation," vol. 1, pp. 246-251 Vol.1, 2004.

[2] C. O'Flynn. "ChipWhisperer Wiki." Internet: https://wiki.newae.com/Main_Page, Oct. 31, 2017 [Nov. 1, 2017]

[3] Kocher, Paul, Jaffe, Joshua, Jun, Benjamin,Rohatgi, Pankaj, "Introduction to differential power analysis," *J Cryptogr Eng Journal of Cryptographic Engineering,* vol. 1, pp. 5-27, 2011.

[4]Data Encryption Standard (DES), National Institute of Standards and Technology (NIST) Std., FIPS-46-3, 1999.

[5] Barenghi A., Breveglieri L., Koren I.,Naccache D., "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proc.IEEE Proceedings of the IEEE,* vol. 100, pp. 3056-3076, 2012.

[6] *Advanced Encryption Standard*, National Institute of Standards and Technology (NIST) Std., FIPS-197, 2001.

[7] Brier, E., Clavier, C.,Olivier, F., "Correlation Power Analysis with a Leakage Model," *Lecture Notes in Computer Science.,* pp. 16-29, 2004.

[8] G. Gogniat, T. Wolf and W. Burleson, "Reconfigurable security primitive for embedded systems," in *2005 International Symposium on System-on-Chip,* 2005, pp. 23-28.

[9] Tiri, K., Verbauwhede, I., "Synthesis of Secure FPGA Implementations," in International Workshop on Logic and Synthesis (IWLS), june 2004, pp. 224–231, Temecula, California, USA