UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Side channel pattern matching using neural networks on FPGAs

Rishab Balaji
M.Sc. Thesis
December 2022

Supervisors:
dr.ir. N. Alachiotis
ir. E. Molenkamp
dr. D.V. Le Viet Duc
ir. D. Vermoen (Riscure)

Computer Architecture for Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Embedded hardware devices like smart cards are prone to threats either due to leakage of information (on a side channel) which could be further analysed to extract important information or it can also be through manipulation of the device functioning by injecting fault from external sources (fault injection attack). Therefore, it is important to develop countermeasures against these threats which require testing using similar attack scenarios. In the context of Side channel analysis and fault injection attacks, a precise triggering device is required to enable the side channel device on the host. While an oscilloscope can achieve this triggering on a selected region of signal, there are certain limitations like existence of random delays/noise in the signal and unnecessarily large measurement windows. For this reason, Riscure has developed an FPGA based triggering device to detect a specific desired reference pattern in the side channel trace and send a trigger signal back to the device under attack/analysis. While the existing design makes use of a Sum of Absolute Differences (SAD) algorithm for the pattern detection, there is scope for improving this pattern matching performance further by reducing number of false positives and negatives and improve latency and resource performance on the FPGA. In recent years, inference of Neural networks on an FPGA has gained popularity and specifically there many instances of low latency inference of such Neural Networks. This project aims to explore the possibility of replacing existing traditional methods and using Neural Networks instead to achieve improved pattern matching results while maintaining similar latency and resource usage targets. Specifically, this project explores low latency inference of Neural Networks on FPGAs. In this thesis, an Multi Layer Perceptron (MLP) is trained with a data-set comprising of a number of side channel traces of which each trace having the occurrence of a selected reference pattern to be detected. Once a network is designed which capable of detecting desired pattern on these side channel traces accurately, further techniques of optimizing the network such that it is feasible for FPGA inference while maintaining low latency and resource constraints are explored. Here, quantization aware training and fixed point design play an important role to achieve FPGA inference with minimal accuracy drop. A design is finalized after experimentation such that it is applicable on various patterns/sizes. This design is then tested on various levels from software testing to RTL simulations to assess the performance on a Xilinx Kintex 7 series FPGA which is the core of the icWaves device. The resulting design achieves a latency of around 430 ns, allows for a maximum size pattern of around 310 samples while running at a maximum clock frequency of 230 Mhz and reports high accuracy.

Contents

Chapter 1

Introduction

This chapter gives the reader a brief introduction of the thesis and also an outline of the report content. The first section of this chapter discusses the motivation behind the thesis project and also the main research questions to be answered. The second section gives a brief outline of the report and its organization.

1.1 Motivation

Side channel and fault injection attacks are serious threats to embedded hardware. These attacks usually require some form of analysis of physical data from the device under attack like power consumption [1], EM (electro-magnetic) waves [2], run times [3] etc during cryptographic operations. Some of these attacks include clock glitching, voltage spiking, optical attacks, etc. For such attacks, the attacker would require a triggering mechanism to enable the side channel acquisition/fault injection device onto the host. The timing of such triggers may need to be accurate depending on the attack scenario. In the past, trigger signals were generated from within the device under test. An example would be the built in triggering functionality in an oscilloscope. However, the side channel signals might contain clock jitters and random delays [4] [5] due to device's counter measures and non deterministic behaviour of some of the programs running on the device. Additionally, the measurement window might be larger than the region of focus resulting in unnecessarily long periods of data acquisition.

To circumvent some of these issues, a pattern based triggering mechanism can be used where an arbitrary region (pattern) on the side channel trace is selected and is then compared with the incoming real time side channel signal to trigger when there is match on the selected pattern. Thus, selecting a suitable pattern matching algorithm as per requirement is very important [6] [7]. The selection of the pattern matching algorithm depends on the following:

- Latency (i.e. pattern-to-trigger delay)
- Maximum reference pattern length
- Number of incorrectly detected/missed patterns (Robustness)
- Flexibility
- Maximum sampling speed (i.e. clock frequency)
- FPGA utilization (i.e. Resource Usage)

Since fault injection may require the timing of the triggers to be precise, having an algorithm which produces low latency and faster response time is beneficial. It is also important to have a robust algorithm so that any inherent noise present in the side channel signal can also be accounted for. Flexibility refers to how well the algorithm can be adapted for the design and testing of the use case which is the FPGA based side channel and fault injection scenario.

Riscure has designed a device namely icWaves (based on a Xilinx Kintex-7 FPGA) for efficient side channel analysis (SCA) and fault injection (FI) testing. This device uses pattern based triggering as its core [8]. A Sum of Absolute Difference (SAD) algorithm is used in icWaves where the trigger is generated when the sum of absolute differences between the selected pattern and the incoming real time side channel signal dips below a specified threshold. It provides a low latency response rate of around 250ns. However the number of false positives and false negatives depends on how high or low the threshold is set.

A similar SCA and FI setup with pattern based triggering as it's core has been proposed by Beckers et al. in [7]. They use a different algorithm named Interval Offset which triggers based on a specified offset value and a threshold value. Their design provides a latency of around 100ns while also providing more flexibility because of their offset parameter which allows the user to set offset over a particular interval where the pattern is to be detected. Again, the number of false positives and false negatives depends on the specified threshold and offset values.

Both these algorithms are simple in nature and can be executed on the FPGA with a fast response time and also within the available logic elements since it only contains addition and difference operations. However, these algorithms are not always accurate and might still contain incorrect detection since real time side channel signals might contain additional noise. Thus, we need to consider robust alternatives that can minimize false positives and false negatives in the presence of the noise while also maintaining response times similar to the above mentioned algorithms which are usually in the range of a few hundred nanoseconds. The algorithm also needs to be efficient in FPGA utilization, maximum possible pattern length, flexibility.

Therefore, this project explores the possibility of a better alternative for the pattern matching algorithm in the icWaves with a good trade off between all the factors discussed with robustness and latency as the main factors.

Neural networks are widely used for pattern detection in real time signals and are known to provide reliable performance [9] [10] [11] [12]. They fit well for the pattern detection problem since side channel signals are also real time signals prone to noise. In recent years, many academic works have implemented Neural networks on the FPGA in various domains. Specifically, a lot of research done recently shows that Quantization of a Neural Networks results in significant decrease in the footprint on the Neural Network and thus low latency results (in range of ns) are possible while still maintaining the accuracy of the network [13] [14]. Building such a Neural network using RTL (Register Transfer Level) on the FPGA would be quite time consuming and also hard to modify after designing since these changes have to be made at a low level. With the advent of high level synthesis, it is possible to design and test such Neural Network design on the FPGA from a high level. Many of the renowned FPGA vendors like Xilinx and Altera have developed advanced HLS tools making it easier to access and make system design decisions at a higher level while creating efficient RTL designs in a minimal amount of time. There is also considerable research being done in recent times towards applying these HLS tools effectively in accordance with deep learning problems to achieve the best possible results [15] [16].

Leveraging the advantages of modern high level synthesis tools and quantized neural networks, this project aims to explore the possibility of using Neural Networks in place of pattern matching algorithms and examine if minimal false positives and false negatives can be achieved while maintaining the latency in ranges similar to that of the previously discussed algorithms like SAD and interval Matching. Trade offs between various factors like pattern size, flexibility, resource usage are also examined.

Therefore, the formulated research questions for this thesis are as follows

1. *How does inferring neural networks on an FPGA for detecting side channel traces compare to existing techniques in terms of accuracy and latency?*

   Since SCA and FI setups need precise and accurate triggering, the primary goal of the project would be to have a design which is both accurate with least number of false positives and false negatives while maintaining a precise low latency design which is comparable to the results from traditional techniques like SAD and Interval matching Algorithms.

2. *What are the possible trade offs in terms of maximum pattern size, resource utilization, clock frequency and flexibility of the design?*

   Further, it is also necessary to understand how the resulting design performs when considering the maximum input pattern size that can be achieved while assessing the clock frequency and resource utilization in comparison to the traditional methods. Design and testing is also an important factor when considering a SCA and FI setup on FPGA as patterns that are being tested needs to be changed often depending on the requirement and therefore it is necessary to consider the flexibility of the design in terms of the workflow involved and their advantages or disadvantages .

1.2 Report organization

Chapter 2 gives the reader some background information on side channel analysis and fault injection along with an introduction of the icWaves design by Riscure.

Chapter 3 is the literature review chapter which initially discusses existing literature on side channel analysis and fault injection based triggering on FPGA followed by further literature exploration on various techniques that could be relevant leading to exploration of Neural Network inference on FPGAs as an alternative.

Chapter 4 discusses the methodology involved in designing and optimizing a Neural Network for the SCA and FI scenario.

Chapter 5 discusses the resulting FPGA design architecture as a result of synthesis of the optimized Neural network using High Level Synthesis.

Chapter 6 discusses the results of the design in terms of accuracy, latency, resource usage, sample size, clock frequency, resource usage and flexibility. This is followed by a brief comparison of these results with the traditional techniques.

Chapter 7 contains the conclusion of the thesis based on the results obtained and hence also answers the research questions.

Chapter 2

Background

This chapter discusses in detail, the background information that is relevant to the thesis. Section 2.1 explains the different types of side-channel analysis and how each type can be used to extract different information followed by the various types of fault injection attacks. Section 2.2 explains the working of the icWaves device by Riscure which is the target device for the resulting design in this thesis.

2.1 Side-channel and fault injection attacks

Embedded devices like smart cards are prone to side-channel leakages and fault injection attacks because they make use of cryptographic operations. These attacks are discussed in [17] which are very different compared to logical attacks and protecting embedded devices against such attacks requires countermeasures at hardware level, kernel level and application level. A side channel attack typically requires physical access to the device under attack to extract information through different physical parameters like time, power and EM radiation. Here, instead of using the intended communication channel/code sections to perform the attack on the device, we exploit the data leaked on a side channel. Smart cards, smartphones and set-top boxes are some examples of devices which are vulnerable to such attacks. These side channels can be exploited in two forms namely,

- Side channel Analysis: A form of passive attack where the adversary listens on a side channel to find important/sensitive information.
- Fault injection: An active attack where the adversary injects faults into the system through a side channel.

2.1.1 Side-channel analysis

As mentioned, side-channel analysis refers to passively listening to a device through a side channel to extract sensitive information. Depending on the operation running on the device, there are two types of analysis techniques to be discussed here,

- **Simple analysis:** Observing the side channel to see if any leakage of information can be captured. Examples of such scenarios include using time, power and EM waves from a device to determine program instructions, signal-to-noise ratio etc. This can be seen in Figure 2.1 where a simple Rivest Shamir Adleman (RSA) encryption algorithm execution shows short and long power peaks which correspond to square and multiply operations respectively. This information can be useful in key retrieval. This is also known as Simple Power Analysis (SPA) or Simple Electromagnetic Analysis (SEMA) based on the measurement made

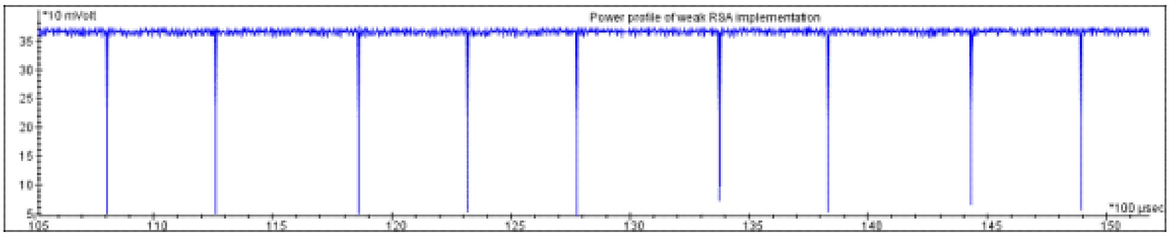


Figure 2.1: Single RSA execution trace for SPA(Source: [17])

- **Differential analysis:** In this type of analysis, the adversary collects numerous traces upon which statistical methods are used to extract information, which might not be possible using SPA/SEMA. An example of such a technique can been seen in the Figure 2.2 where several traces are acquired and divided into two groups based on one bit in the intermediate data. These traces are further processed using mean differences to obtain a differential trace. This differential trace will provide more information on the intermediate data.

2.1.2 Fault injection

In contrast to side channel analysis, fault injection aims to interrupt a defined program flow or to change important values during execution. Such attacks may include allowing to skip authentication process, bypassing an unauthorized firmware or security measures. Fault injection can be achieved through several different ways namely:

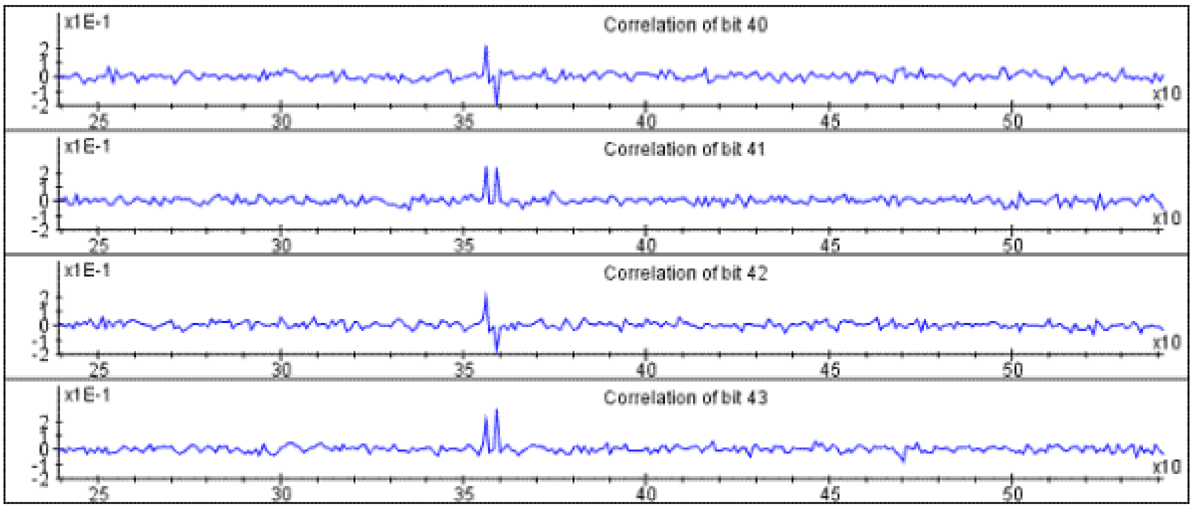


Figure 2.2: DPA on multiple RSA traces to reveal secret key(Source: [17])

- **Voltage/Power glitching:** Disrupting the power supply to the storage/memory modules at specific points such that the read values to the memory are incorrect.
- **Clock glitching:** In this technique, short clock pulses are injected to vary the internal clock of the device. This in turn affects the operation being executed on the device. By timing these pulses precisely, one can skip instructions and bypass security measures in different cryptographic operations.
- **Optical glitching:** A circuit level attack where laser light is used for transistor switching. Such low level attacks can not only affect memory containing instructions to break cyptographic executions, but can also have catastrophic outcomes where the microcontroller is reset, EEPROMs and flash memories are affected.

2.2 icWaves

Embedded devices need to be properly protected from some of the passive/active attacks discussed previously in section 2.1, by deploying countermeasures. For this purpose, Riscure has developed a device namely icWaves (figure 2.3) for embedded security testing purpose. This is an FPGA-based triggering device that uses real time pattern detection to provide accurate triggering since the timing of the attacks is very important. Conventionally, reset/communication signals is used where the device's built-in timer can be used to send a trigger pulse. However, due to inaccuracies like jitters/drifting in the clock and also random program (running on the

Figure 2.3: icWaves device(Source: [8])

device) interrupts it may be hard to achieve accurate triggering. When performing a Side channel analysis, we might also run into the issue of unnecessarily large measurement windows which might result in longer periods of data acquisition. To combat this, we need a device to detect a selected reference pattern just before the measurement or injection. Because of these reasons, icWaves employs pattern based triggering as its core.

Figure 2.4 shows the internal blocks of the icWaves FPGA device. It consists of blocks to acquire and store a desired reference pattern where measurement/fault injection is to be performed. Figure 2.5 shows the pattern detection process where the selected reference pattern is then compared with the incoming real time EM wave or power signal (after A/D conversion) in the Waveform matching module to produce a trigger signal to the embedded device under test. The waveform matching module inside the icWaves uses a Sum of Absolute Differences algorithm which compares the incoming real time signal with the acquired reference trace using a sliding window approach where window size is based on the chosen reference pattern size. A sum of the absolute differences between the two signals is computed over the window and if the result drops below a predefined threshold value, a trigger pulse is produced. The remaining blocks correspond to sampling the input signal, compensation for noise, control logic for the overall device etc. This device consists of a Xilinx Kintex-7(xc7k160t) FPGA device. It stores and detects either a reference pattern of 1024 samples or 2 reference patterns of 512 samples size. It has a pattern-to-trigger delay of 250 nanoseconds with a configurable hold off/delay time. It has a sampling speed of 200 MSamples/sec with each input sample having 8 bit precision. While designing icWaves, the most important aspects are latency and accuracy of the results since the triggering needs to exact and precise. However, resource utilization and sampling speed are also important factors to consider with respect to the target FPGA device.

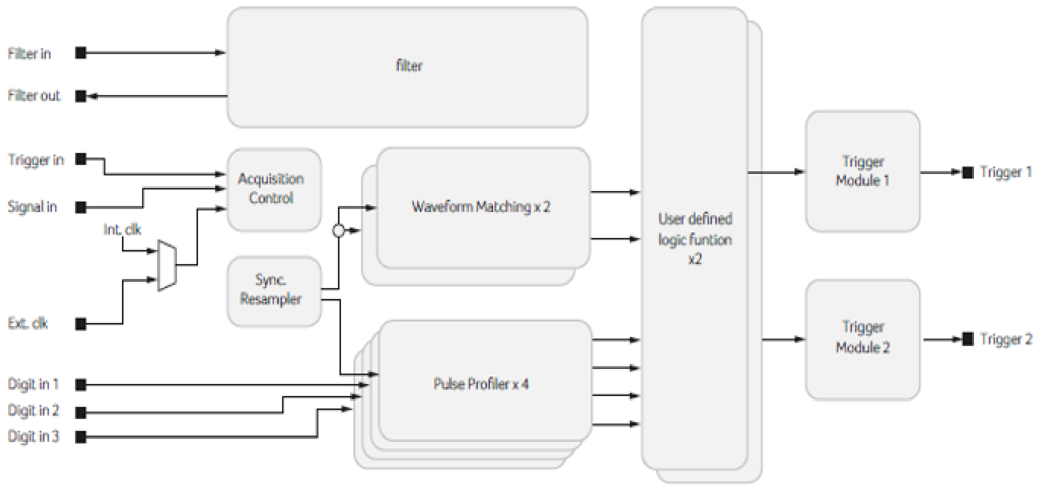


Figure 2.4: Internal blocks of icWaves (Source: [8])

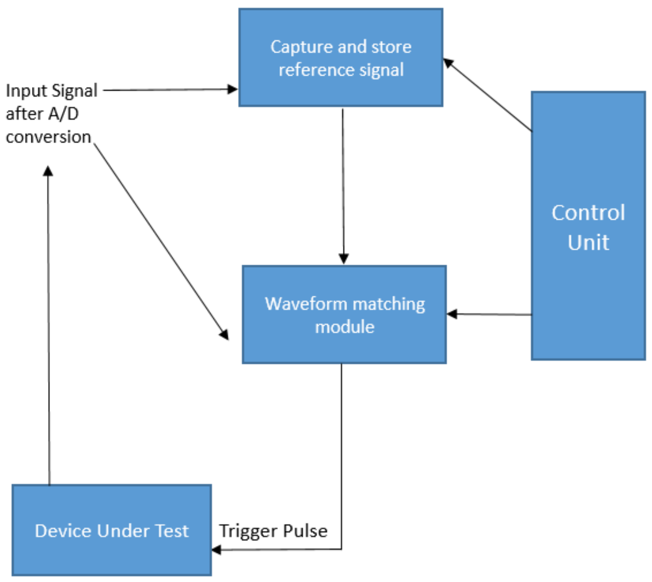

Input Signal after A/D conversion
Capture and store reference signal
Control Unit
Waveform matching module
Device Under Test
Trigger Pulse

Figure 2.5: Pattern detection in icWaves

Chapter 3

Relevant Work

This section discusses the relevant work in regards to traditional pattern/waveform matching techniques used in SCA and FI testing while also discussing the need for more exploration. Further, this section explores alternate solutions for pattern matching and signal similarity by examining existing literature from various domain applications to see if it can be applied here.

Side channel analysis and fault injection is executed by sending a trigger signal from the host device to the embedded device at a certain region of interest. As already discussed in section 1.1, triggering using the host device's built-in triggering mechanism can sometimes be ineffective and thus there is a need for an improved pattern based triggering mechanism.

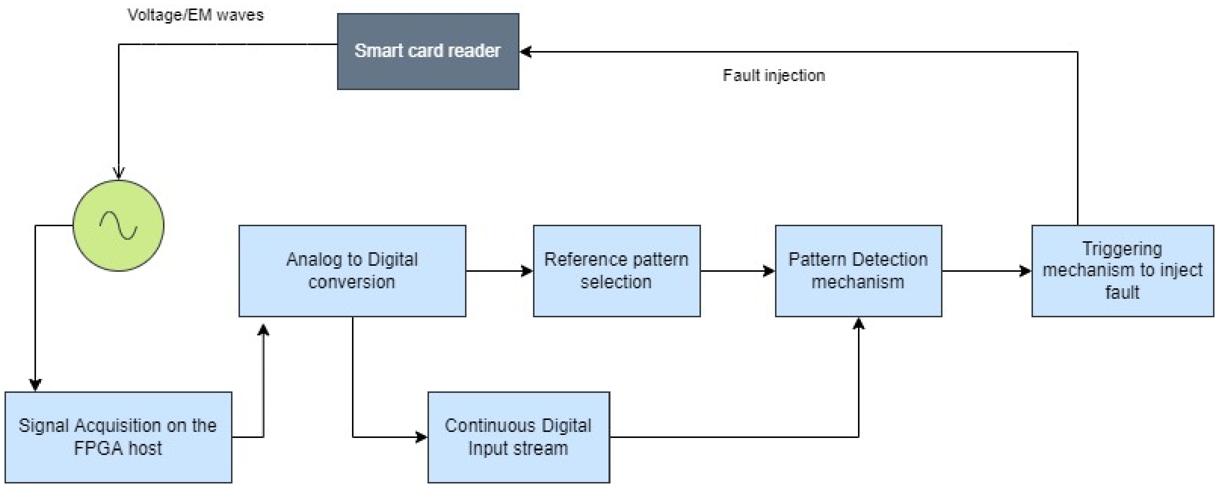

Voltage/EM waves
Smart card reader
Fault injection
Analog to Digital conversion
Reference pattern selection
Pattern Detection mechanism
Triggering mechanism to inject fault
Signal Acquisition on the FPGA host
Continuous Digital Input stream

Figure 3.1: FPGA based setup for SCA/FI using pattern detection

A pattern based triggering requires using algorithms that can detect an arbitrary region on an incoming side channel signal and send a trigger signal instantaneously. A low latency triggering is very necessary since the attacker does not want to miss the window to intervene the system. Traditionally, these setups are built using FPGA devices as they provide the best accuracy and precision while allowing modification of the system design since through reconfiguration. The setup required for such Side channel Analysis and Fault injection on a Smart card reader can be as seen in the Figure 3.1.

In the past, academic works have considered algorithms which can provide signal/pattern similarity as the most suitable techniques. One of these is the Sum of Absolute differences (SAD) algorithm which is currently employed on the icWaves device by Riscure (discussed in section2.2). Its working has been explained in [6] where the pattern detection is implemented by comparing the incoming real time side channel signal with a desired reference region of interest. The difference between the samples are compared and then the absolute values are added to find the sum. If the sum is over a specified threshold, then a match of the reference pattern is found on the incoming signal and thus a trigger signal is sent. This setup runs at 200 MHz with a latency of 250 ns and can detect pattern of up to 1024 sample length. The amount of false negatives and false positives is decided on the specified threshold value.

In addition to the Sum of absolute differences algorithm, there are other alternatives to achieve similar results. The authors of [7] also discuss a similar triggering system for side channel detection and fault injection. They make use of an algorithm namely interval matching, where an interval is defined both above and below the reference pattern. The output trigger defined is based on whether the incoming signal lies inside this interval. This interval can be configured using an offset such that it allows for flexible pattern detection. They achieve latency of 128 ns with a maximum sample length of 2625 samples with 125 MHz sampling rate.

Both the SAD and interval matching algorithm compare incoming signals using a moving average concept with the a static window over the reference pattern with shifting incoming samples from the real time signal. Further, the authors of [7] also explore the feasibility of cross-correlation algorithm using the same moving window principle where the samples are multiplied and their product determines the similarity between the two signals. However, multipliers are required to implement this algorithm on the FPGA. Since the number of DSP blocks on the FPGA is limited, these multiplication operations can be harder to accommodate if the size of the reference pattern window keeps increasing. So, the authors discard the use of this algorithm for the SCA and FI based triggering.

The cross-correlation algorithm discussed previously is conventionally used in signal processing applications and is often a very reliable measure of signal similarity. The authors of [7] already address issues with cross correlation and the need for increased multipliers with increase in pattern length. However, this length can be significantly reduced if converted onto the frequency domain. Frequency domain cross correlation for signal similarity on FPGAs have been discussed in [18], [19] and [20]. The authors of [20] address the issues with time domain cross correlation by comparing serial and parallel implementations. Using a serial implementation provides a low latency and high processing speed, but hardware resource consumption also increases linearly with increasing in the sample size. Using a parallel architecture, reduces the resource consumption significantly but also affects latency negatively. Based on these results, they move to an FFT based solution where the cross correlation is computed in a frequency domain. Here FFT IP cores with best performance are used to convert the time domain signal to frequency domain and the cross correlation is then computed. Once converted to frequency domain, the sample size reduces comparatively. The resulting signal after correlation is converted back to time domain using an inverted FFT operation. The architecture utilizes BRAM and DSP blocks. The number of DSP blocks (multipliers) used is independent of the sample size since BRAM is used to store data and acts as a buffer. But the BRAM usage increases linearly with the sample size.

A solution that can provide a latency in the range of a few hundred nanoseconds is needed while the results of FFT cross correlation are in milliseconds range. This delay is due to the use of Block RAMs. Since the project at hand only deals with small amount of samples at the input array, using BRAM addressing (used for large amounts of data) for storing input data can be comparatively slower as opposed to using logic blocks to store the input array which can result in increased clock cycle delay. Since this method hinders the latency demand, it is not considered to be feasible.

Another possible alternative to signal similarity algorithms is to consider string matching([21] and [22]) where each sample from the reference trace can be matched with each of the samples from the incoming sample. Knut-Morris-Pratt algorithm is one of the most efficient string matching algorithm which is discussed in [23], where the authors present an Finite State Machine using the on chip RAM to provide an optimized implementation of the string matching on an FPGA. However, this algorithm works on the principle of one-to-one matching where if a match is not found, the memory stores this information after which the FSM uses this to optimize the process and reconfigure itself. This algorithm hinders the use case as real time side channel signals are used where reference pattern and the incoming real time signal might not have the same exact pattern but still be similar. Another similar implementation can be found in the form of Dynamic Time Warping technique found in [24]. These could be considered since they provide an approximate similarity measure but all these algorithms need to store large amounts of data which would again require use of Block RAMs or similar memories whose addressing is expensive in terms of clock cycle delays resulting in much higher latency(range of millisecond or higher) and would be infeasible considering the trade offs.

Since side channel signals are real time signals containing noise, Neural Networks models can prove to be extremely effective. Specifically for side channel and fault injection attacks, recent research shows that Deep Neural Networks can now bypass all the countermeasures (like clock jitters, misalignment and program interrupts) that were previously employed to prevent leakage due to traditional attack methods [9]. The authors of [12] discuss how Neural Network models should be assessed for side channel analysis. They discuss how the points of interest (leakage data) in side channel signals can be captured effectively by properly choosing the hyper-parameters. They also explore how regularization can help prevent over fitting and learn from leakage samples effectively to achieve precise results. Regularization is technique used to avoid over-fitting problems in Neural networks where certain weights having large values are penalized and therefore makes the weights to be sparsely distributed.

It has been established, that neural networks can be used to achieve accurate results when applied to SCA and FI domain. However, inferring these Neural networks on an FPGA device poses certain challenges like accuracy drop after inference, FPGA resource limitations and latency constraints. In recent years, there has been a significant amount of research done in achieving inference of Neural network while maintaining the accuracy, resource usage and latency demands. Specifically, research shows that quantization of Neural networks can help reduce the footprint of Neural networks significantly allowing for feasible inference on FPGAs.

Inference of neural networks on FPGAs requires designing the architecture using digital logic and circuits using low level languages like VHDL or Verilog while having data represented in fixed data type. Many previous academic works have discussed the implementation of such a NN on FPGA exploring the procedure of building each neuron, activation functions etc. Each of these have their own implications on the hardware performance in terms of LUTs used, memory required, delay caused etc. In recent times, the use of high level synthesis to realize hardware designs on the FPGA has gained a lot of popularity since these tools allow the user to design the architecture at a high level which is further translated into low level RTL code and synthesized to obtain the desired optimum result on the desired target FPGA. These tools are highly sophisticated with inbuilt features (like loop unrolling, data-flow transformation techniques) to obtain the most efficient result [25]. This allows for customization of the neural network as needed at a higher level without modifying the final RTL design. The authors of [13] discuss a framework where such high level synthesis tools are utilized in inferring a deep neural network for a particle physics application which has extremely low latency demands to achieve inference in the range of 100 ns latency on an FPGA device. The authors outline the benefits of designing the network network at a high level and provide a means of control at the high level after which the network footprint is significantly reduced by means of compression, quantization and paralellization. This makes the inference of the network feasible on the FPGA device while still keeping the accuracy and performance intact. The results also portray how the compressed network reduces the resource usage significantly while performance remains unaltered. Further, the authors of [14] also discuss a similar framework designed specifically for binary neural networks and quantized neural networks with up to 1-bit weights and activation where these outputs can only have a value of +1 or -1 with set bit representing +1 and unset bit representing -1. These networks are mostly tested on Image Neural Networks and verified to produce same level of sub-microsecond to nanosecond performance. The core concept behind these successful inferences with low latencies and resource usage is called Quantization. It refers to the process of converting the floating point values into fixed point equivalent lowering the precision, effectively reducing the size of the network such that it can satisfy the FPGA hardware requirements. The authors of [26] discuss two different types of Quantization namely Post-Training Quantization and Quantization-Aware Training. They discuss the implication of both these techniques of precision reduction and summarize that Quantization Aware Training is more superior since it has better retention of the overall accuracy of the network in comparison to post training quantization. This is due to the fact that the quantization process is also linked to the training parameters of the Neural Network. There are different frameworks and Quantization libraries which assist this process making it easier to incorporate this concept into the Neural networks at a higher level [15] [16].

Therefore, the state of the art techniques like SAD and Interval matching provide low latency designs while allowing a maximum of around 1000-3000 samples. These are not precise algorithms since they measure similarity by measuring the average of a sliding window. Their accuracy depends on algorithmic parameters like threshold and offsets which decide the number of false positives and false negatives.

Cross correlation on time domain using a similar sliding window approach suffers from hardware resource issues while FFT based frequency domain correlation suffers from latency issues.

String matching algorithms also suffer from latency issues since they need large amounts of data to be stored and accessed, increasing the overall clock cycle delay.

However, research on Neural Networks inference on FPGAs report low latency results (in ns range) as required while also satisfying resource constraint. Neural Networks in general seems like an efficient solution over traditional algorithms to achieve improved matching and detection accuracy with comparatively less false positives and false negatives since real time signals are involved. These networks can be made extremely accurate if trained with side channel signals and appropriate information on the desired pattern occurrences. In combination with modern HLS tools, Neural Network design testing and modification can be made simpler.

Chapter 4

Methodology

This chapter discusses in detail, the methodology involved in designing a neural network architecture with a set of side channel traces as the input data to detect a specified arbitrary pattern in each of these traces. This network is then optimized for FPGA inference by using techniques like quantization and regularization. The entire methodology can be summarized as follows:

1. **Side channel power traces and data-set creation:** This section deals with the procedure of converting a set of side channel traces into equivalent data set based on a desired arbitrary pattern to be detected in each of these traces. Specifically, this process involves preprocessing of data by collecting the occurrences of the pattern in each of these traces using a correlation algorithm. The trace set along with the pattern occurrence information forms the data set, which is then labelled using the pattern occurrence information.

2. **Neural Network Architecture:** Next, a neural network architecture is designed for detecting the pattern on the basis of the data-set generated. This section explores the preprocessing of the data-set like scaling/normalization and regularization. The Neural Network model is then tested for various metrics like accuracy, precision and recall etc to achieve the optimum network design.

3. **Quantization and Fixed-point representation:** This section explores how the footprint of the neural network can be reduced such that it can fit on the target FPGA by limiting the resolution of model parameters like weights, bias and activation. This process is known as Quantization. This section also explores the extent up to which the precision of the model can be reduced without significant accuracy drops.

4.1 Side-channel power traces and Data-set creation

The first step before building a Neural network is to have a data set on which the network can be trained and tested upon. This section explores in detail how a set of side channel traces are converted into equivalent data set and further labelled based on the occurrence of a desired arbitrary pattern to be detected.

Figure 4.1 shows a side channel power ***trace***. Such a trace usually represents various operations running on the embedded device. For a side channel attack, the user needs to select an arbitrary region along the trace where an attack is desired. This region defines the ***pattern*** to be detected. In general, it is always better to select patterns which are unique since the trace might contain repetitive peaks such that false detection can be avoided. Having patterns of large lengths are also advantageous since it can incorporate more features improving detection quality. While selecting such a trace, adding a small amount of noise can also help improve the detection if the real time side channel signal contain inherent noise.

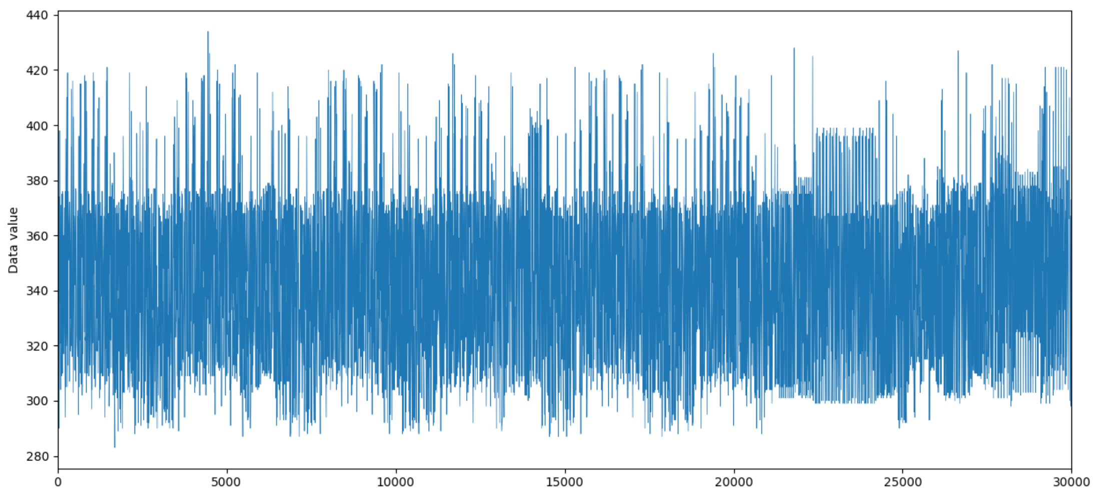


Figure 4.1: A side channel power trace

As already discussed previously, embedded devices might contain built-in countermeasures to combat side channel attacks. One such countermeasure is to introduce misalignment into the traces while operations are running on the embedded device. Therefore, a *trace set consisting of 10* such misaligned traces are considered for the creation of data set. Each of these traces are representative of repetitive operations running on the embedded device. But the collected traces might not directly have this information since it contains misalignment from the original trace. A desired pattern might occur in each of these traces but might be misaligned by a number of samples.

To make it easier for the Neural Network to learn these traces(supervised learning), a preprocessing step is introduced where a cross correlation algorithm is used in software to detect an arbitrary region (pattern) from the original trace along the remaining 9 misaligned traces. This can be visualized in Table 4.1 where in each of these traces, the occurrence of the desired reference pattern is calculated and the location is decided based on the location of highest correlation factor.

Trace	**Location**	**Correlation**
1	563	0.980456
2	462	0.970324
3	552	0.993632
4	204	0.985791
5	428	0.985384
6	-4727	0.991842
7	867	0.991239
8	337	0.987978
9	229	0.990847

Table 4.1: Reference pattern occurrences

The 10 misaligned traces form the data set. Each of these traces consists of 30,000 samples. Each of these samples are annotated using labels namely ”1” or ”0” to assist the neural network to detect the pattern of interest. This annotation is done on the basis of the pattern occurrence data which has already been calculated as seen in 4.1. All the samples along a reference pattern of length n are labelled ”1” starting from its occurrence location (from 4.1) in the respective sample. The remaining samples of the (non pattern occurrence regions) will be labelled with a ”0” label.

Finally, the data set will be split where 70 percent (7 traces) are used for training of the network and remaining 30 percent (3 traces) are used to test the network performance. The entire data set creation and labelling process can be visualized as seen in 4.2.

4.2 Neural Network Architecture and optimizations

Once the data set has been created using the side channel traces, a suitable neural network model is to be designed for a binary classification since the data set contains two labels namely ”1” and ”0”.

An MLP (multi layer perceptron) model is considered here. To ensure least possible footprint, the smallest possible model with least amount of Neurons and layers

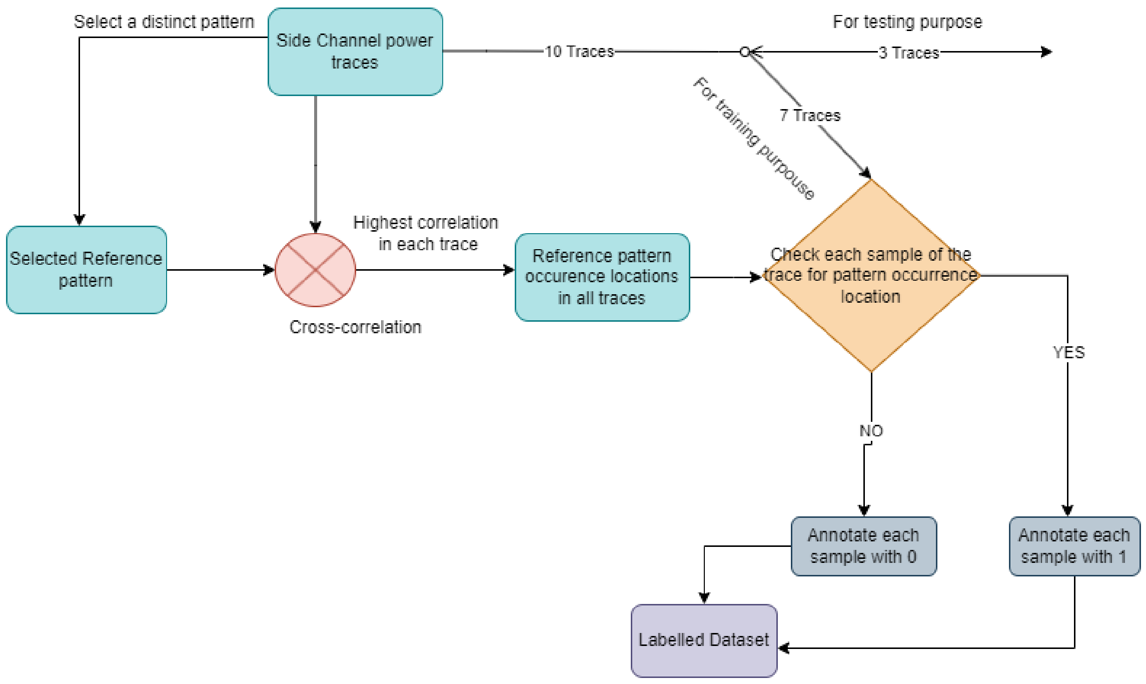

Select a distinct pattern
Side Channel power traces
10 Traces
For testing purpose
3 Traces
For training purpouse
7 Traces
Selected Reference pattern
Highest correlation in each trace
Cross-correlation
Reference pattern occurence locations in all traces
Check each sample of the trace for pattern occurrence location
YES
NO
Annotate each sample with 0
Annotate each sample with 1
Labelled Dataset

Figure 4.2: Converting side channel traces into equivalent data-set

are considered since it will further be mapped on to the FPGA. All the layers of the MLP will be dense layers where each neuron of the layer is connected each of the neurons in the preceding layer. Sigmoid activation ([27]) is used in the output layer since the resulting output will be a 0 or 1 which is representative of whether a pattern has occurred or not. A Rectified linear activation function ([27]) is used in the intermediate layers since it widely popular for easier training and better performance. Other data preprocessing techniques like scaling and normalization are also considered for improved results.

4.2.1 Quantization for FPGA inference

Once the network architecture is designed, it needs to be mapped on to the FPGA. However, the network by default will consist of weights, bias, input and output values in floating point representation. These floating point values will need a very large amount of bits when mapped on to the FPGA. However,this can be hard to accommodate since the FPGA has limited hardware resources.

$$x_n = g_n(W_{n,n-1} * x_{n-1} + b_n) \tag{4.1}$$

Equation 4.1 portrays the operation that occurs inside each neuron of the network. Here x_n is the output from each neuron, g_n is the activation function, $W_{n,n-1}$ is the incoming weight from the previous neuron and b_n is the bias value. When such a neuron is to be mapped onto the FPGA hardware, it is important to know what parts of the FPGA will be utilized based on the operation. For the addition operation, logic cells of the FPGA will be used. Since there is a multiplication involved, DSPs may also be utilized. The activation functions are pre-calculated and stored in memory. DSPs are very limited even on modern FPGAs. The logic cells and even large memory storages like BRAMs also cannot handle large floating units as they require large number of bits to represent on hardware. To overcome this issue, the footprint of the designed Neural network and its values needs to minimized ensure successful mapping on the FPGA.

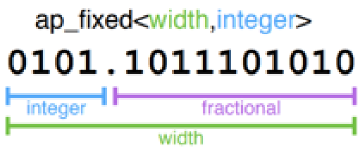


Figure 4.3: Fixed point representation(Source: [28])

As already discussed in Chapter 3, many literary works suggest the process of Quantization to successfully overcome the floating point issues and achieve neural network inference on the FPGA. Quantization is the process of converting large values into discrete fixed size values. Research also suggests using Quantization Aware training over post training quantization as it retains the overall accuracy much better even after the quantization process. The idea behind Quantization Aware training is to make the network aware of the desired/specified quantization during the training process as opposed to after the training (in case of post training quantization). This is done by introducing functions known as Quantizers into all the layers of the network. The Quantizers are similar to activation functions of the Neural Network, where the quantizer maintain the precision of the incoming values based on the requirement by using fixed point values in place of the existing floating point ones. The Quantizer function only tags the values during the forward pass of the training iteration. This mechanism allows for a network whose footprint is minimized according to the specified quantization making it feasible for FPGA inference while also having minimal accuracy loss during the conversion. Figure 4.3 shows the representation of the values in fixed point as should be used for the resulting RTL design as opposed to floating point values. Here it can be seen that the amount of bits used to represent the integer part and the fractional part of a value needs to be specified separately.

4.2.2 Fixed point tuning and Regularization

As the network undergoes quantization aware training, the values are still represented in floating point representation. However, these values are lowered in precision (due to use of Quantizer functions in the network) which means that it can now fit inside equivalent fixed point values if converted. Therefore, the next step is to select the fixed point precision for the weights, bias, input and output values based on the range in which they occur. Previously, there was a discussion on how using quantized lowered precision values can reduce the hardware footprint while still maintaining the accuracy. However, there is also a chance of ending up with completely inaccurate results if the fixed point precision to represent this quantized data is not chosen properly. Hence, it is to be ensured that the network's input and output values have appropriate and sufficient level of precision to be represented and further might also need to tune the precision of values in the intermediate layers like the weights, bias and activation function outputs according to the quantization specified during the training such that the least possible precision can be used to represent these values while also having the best accuracy.

This process of assigning fixed point precision can be started off by using a default fixed point precision for the entire model (all values including weights,bias and IO). This is followed by plotting a graph to check the range over which different values are distributed against the fixed point precision range. This graph is useful to understand whether the selected precision range is sufficient to represent the equivalent value. If not, the value might be falsely represented leading to inaccurate result. During the process of assigning the precision, an extra bit is always added so that the sign of the value (+ve or -ve) can also be accounted for. This sign bit ensures that there are no rounding off and saturation errors. If the assigned precision is larger than required, it can further be reduced and tuned accordingly. Analyzing the information from the graph, all the weights, bias,input and output values are further tuned such that least possible precision is assigned to each of them while maintaining the highest overall accuracy. This might also result in going back and changing the quantization specified for the network (during quantization aware training) and retraining the model again if necessary.

Once the quantization for the network values are finalized and the equivalent precision is selected, there is also a need to make sure that the resulting architecture is universal (i.e same model should work similarly if input pattern or size changes). This is really important to consider because a change in input will also change the overall weights and bias values which might also affect the selected quantization and fixed point precision. If the model cannot account for these changes, it may provide inaccurate results upon change in input parameters. Therefore, it is necessary to ensure the stability of the model created. To overcome this issue, regularization can be introduced during the training of the Neural Network. Specifically, L1 regularization ([27]) is used here which adds a penalty in each layer for large weights during the training process. Due to this penalty, it is ensured that the weights are not scattered over different ranges. This weight sparsity makes selecting the precision easier since they occur over a shorter range over the precision plane.

This concludes the methodology involved in building a neural network architecture with side channel traces as input data set such that it can be made feasible for inference on the FPGA hardware. The next step is to convert this model into equivalent RTL architecture design using High Level Synthesis tools and also analyze its results.

Chapter 5

Network Architecture and Implementation

This chapter discusses in detail the network architecture design that is obtained after experimenting and also tuning of the fixed point precision values to achieve an optimum solution for the FPGA inference.

Before moving to the actual design, it is also important to know the different software and hardware platforms that were utilized in achieving this result.

- The Neural Network designed here is intended for the pattern matching module inside the iCWaves (section 2.2) SCA and FI device designed by Riscure. It consists of a Xilinx Kintex 7 series FPGA ([29]). The icWaves pattern matching module allows data of 8 bit precision and therefore the network designed will be using 8 bit as the input data precision.
- The dataset for the neural network is created using a set of 10 misaligned side channel traces provided by Riscure.
- For developing the neural network architecture, TensorFlow is used along with Keras ([27]).
- For quantization aware training, Qkeras library ([30]) is used which acts as an extension to assist in Quantization of the network during training.
- For tuning the necessary fixed point precision values and further converting the network into optimized C code for HLS, a framework namely hls4ml ([28]) is used.
- Finally to convert the network from C to required RTL specification (Verilog or VHDL) for synthesis on the target FPGA, Xilinx's Vitis HLS Tool ([25]) is used.

The design follows a top-down process where ideas are first designed and tested on a higher layer and then the details and issues for a low level design are explored.

Initially, a reference pattern is selected for detection from the set of misaligned side channel traces collected. The labelled data set is created using the pattern occurrence locations on these traces as seen in table 4.1. These values might differ based on the pattern selected. To imitate a real time signal, a small amount of Gaussian noise might also be added which further can help the network for better detection. Since the matching module of icWaves has 8 bit input interface, the designed network must also work on the same kind of input. By default, the values from the side channel traces are in 16 bit representation. Therefore before training, the labelled data set and all its values are converted into equivalent 8 bit signed integer. This is done using the formula,

$$X_8 = ((X - min(x))/(max(X) - min(X))) * 255 - 128 \tag{5.1}$$

where X_8 is the 8 bit result, X is the input value and min and max are minimum and maximum values of the data set(in 16 bit) respectively.

Figure 5.1 shows the architecture of the Neural Network that was finalized after experimentation. The first layer is an input layer namely ”fc1 input” and it accepts input of sample size equal to the selected reference pattern (154 in the case of Figure 5.1) to be detected in 8 bit representation. This is followed by two dense layers namely ”fc1” and ”fc2” consisting of 12 and 6 neurons respectively along with their respective Rectified linear (ReLu) activation functions followed by the output layer which consists of a single neuron followed by sigmoid activation which gives the final output of the network. Since quantization aware training is used during the training of the network, the weights, bias, input and output values of all the layers are maintained to remain below a specified precision. Here, the selected precision during quantization aware training is 15 bits for representing the entire value with 7 bits for the integer representation and remaining 8 bits for the decimal representation making it possible to represent these values using fixed point equivalent. The same can be seen in all these layers where ap fixed is the fixed point precision assigned for representing these values in the HLS C code with 16 bits to represent the whole value with 8 bits for integer and 8 bits for decimal. It can be seen that an extra bit has been added while assigning fixed point representation in HLS C. This is to compensate the sign bit as discussed in section 4.2.2.

To achieve this design architecture, several optimizations and changes were involved before reaching the final version. Before introducing any kind of optimization techniques (like quantization and regularization) that would ease the network mapping on the FPGA, a basic network architecture consisting of three layers (one input, one hidden intermediate and one output layer) with minimal architecture of 8,4 and 1 neurons in the first, second and the output layer respectively were considered.

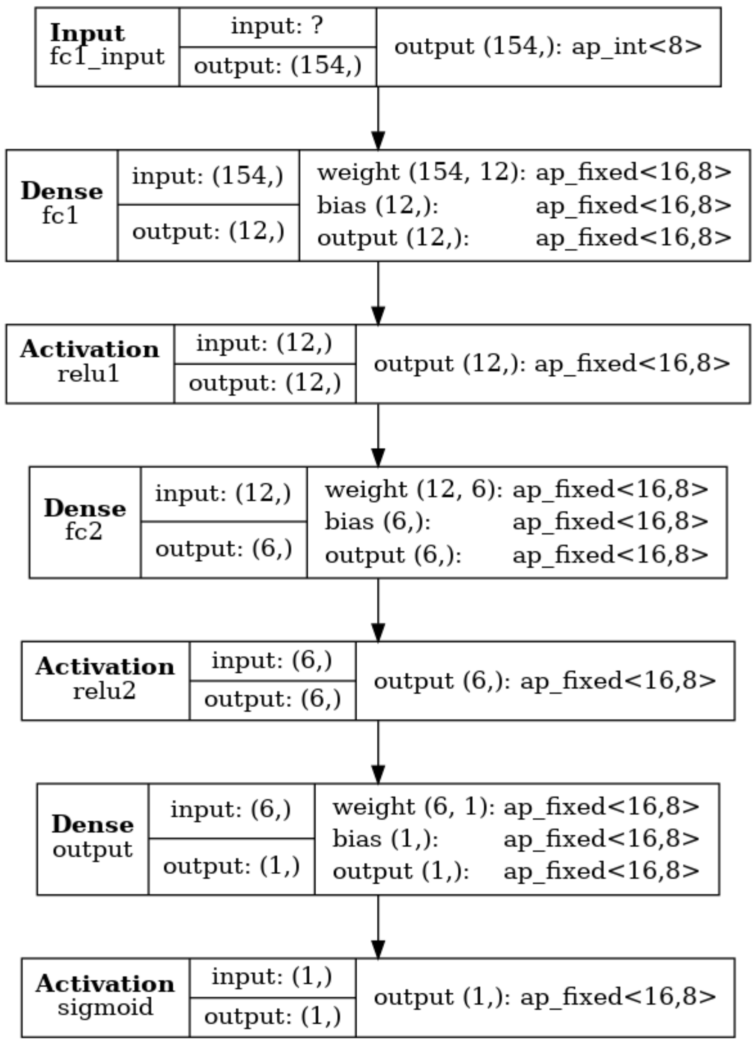
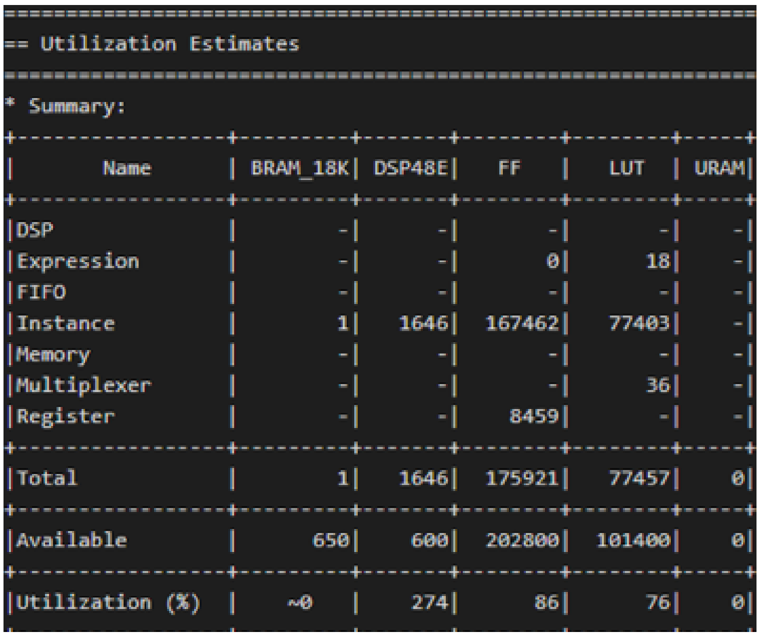

Input
fc1_input
input: ?
output: (154,)
output (154,): ap_int<8>
Dense
fc1
input: (154,)
output: (12,)
weight (154, 12): ap_fixed<16,8>
bias (12,): ap_fixed<16,8>
output (12,): ap_fixed<16,8>
Activation
relu1
input: (12,)
output: (12,)
output (12,): ap_fixed<16,8>
Dense
fc2
input: (12,)
output: (6,)
weight (12, 6): ap_fixed<16,8>
bias (6,): ap_fixed<16,8>
output (6,): ap_fixed<16,8>
Activation
relu2
input: (6,)
output: (6,)
output (6,): ap_fixed<16,8>
Dense
output
input: (6,)
output: (1,)
weight (6, 1): ap_fixed<16,8>
bias (1,): ap_fixed<16,8>
output (1,): ap_fixed<16,8>
Activation
sigmoid
input: (1,)
output: (1,)
output (1,): ap_fixed<16,8>

Figure 5.1: Finalized Neural Network design

Since the end goal is to achieve a network architecture that could be successfully mapped onto the FPGA, smallest possible network is considered to reduce network size as much as possible. Going any lower than the specified combination would result in inaccuracies. Initially when testing the network performance for accuracy, it was found that this architecture would work for certain input patterns and fail for others pattern variations. This issue was solved by experimenting with preprocessing techniques like scaling and normalization of the input data. However, introducing such preprocessing of data also meant that deploying similar mechanism for the input stream of real time data on the FPGA based side channel device which would require additional processing elements and also memory storage and accessing. This would lead to increased resource utilization and latency demands which was not favourable. By further increasing the number of neurons and further reducing the learning rate, it was found that these scaling and normalization techniques were no longer required and the network worked on the raw incoming data.

Once the network was deemed accurate (in terms of false positive and false negatives) on a software level, the next step in the process was to map this onto the FPGA. For this, equivalent HLS C code suitable for High Level Synthesis by the Xilinx tool would be required. Here, a tool namely hls4ml for is utilized to convert the Neural Network design (written in python) to equivalent HLS code code along with specifying the desired target FPGA (Xilinx Kintex 7 in this case) and the target clock frequency (200MHz in this case). In the first attempt of trying to map the network onto the FPGA, it was found that the resulting design would have an extremely high resource utilization due to the presence of floating point values in the network. This can be visualized as seen in the Figure 5.2. To reduce this high resource utilization, optimization is required which improves the resource utilization drastically (which will be discussed in figure 6.8 of chapter 6).



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+--------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |   LUT  | URAM|
+-----------------+---------+-------+--------+--------+-----+
|DSP              |        -|      -|       -|       -|    -|
|Expression       |        -|      -|       0|      18|    -|
|FIFO             |        -|      -|       -|       -|    -|
|Instance         |        1|   1646|  167462|   77403|    -|
|Memory           |        -|      -|       -|       -|    -|
|Multiplexer      |        -|      -|       -|      36|    -|
|Register         |        -|      -|    8459|       -|    -|
+-----------------+---------+-------+--------+--------+-----+
|Total            |        1|   1646|  175921|   77457|    0|
+-----------------+---------+-------+--------+--------+-----+
|Available        |      650|    600|  202800|  101400|    0|
+-----------------+---------+-------+--------+--------+-----+
|Utilization (%)  |   ~0    |    274|      86|      76|    0|
```

Figure 5.2: FPGA Resource utilization report for a network without quantization

Therefore, Quantization aware training is introduced to overcome this FPGA mapping issue. For this, a library named Qkeras ([30]) is used which is extension to Keras framework on which the Neural Network is built on. Qkeras replaces the dense and activation layers of the network with equivalent layers namely QDense and QActivation which are layers with quantization function. As discussed in section 4.2.1, Quantization aware training involves specifying a desired precision of quantization for weights, bias, input and output values on which the network will be trained upon. This process is initiated by trying to limit the values to a fixed point precision of 8 bits where 4 bits are allocated for the integer part and the remaining 4 bits for the fractional part. However, this results in an inaccurate solution. On analyzing the values of the different weights of the network, it was seen that these values need more bits to accommodate the entire information. Further, it was also seen that the fractional part required more bits for representation than the integer part. Several of these quantization variation were experimented upon to achieve accurate result for a specified pattern. However, it was observed that changing the input parameters on which the network is trained upon like the input pattern would also require changing the specified fixed point precision accordingly. This is because of the variation in the resulting weight values. To overcome this issue, L1 regularization ([27]) is introduced into the network which in turn reduced the range over which the resulting weights values would occur and therefore keeping all the weights in a sparse range rather than scattered. This ensured that the specified network architecture with the desired fixed point precision would not be affected regardless of the input parameter variations.

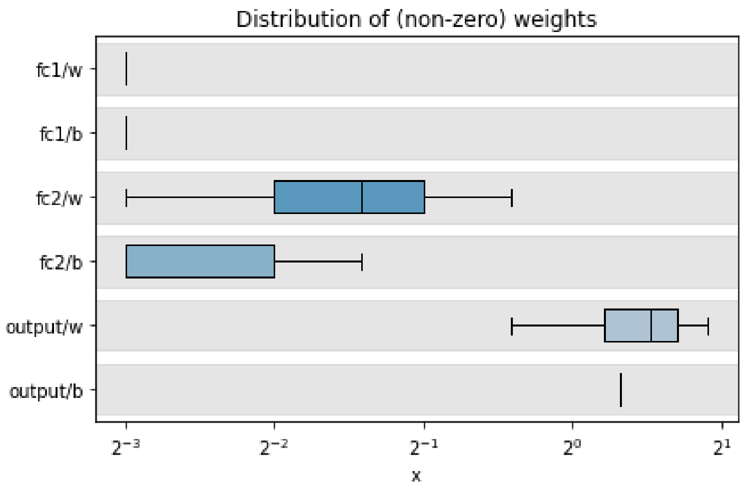


Figure 5.3: Before optimization

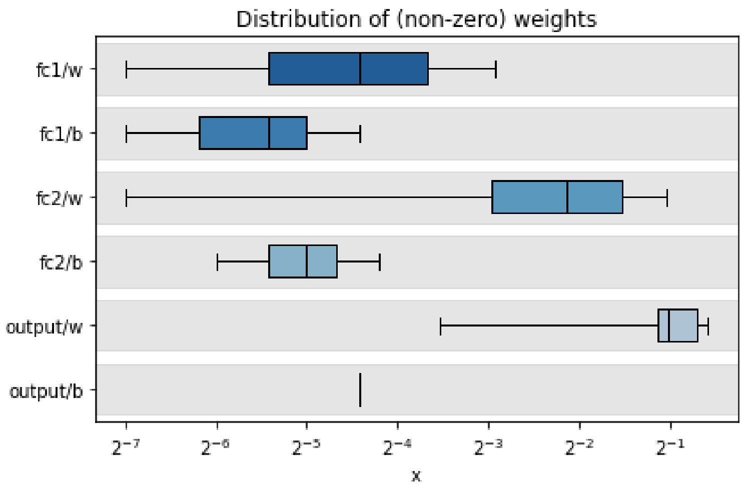


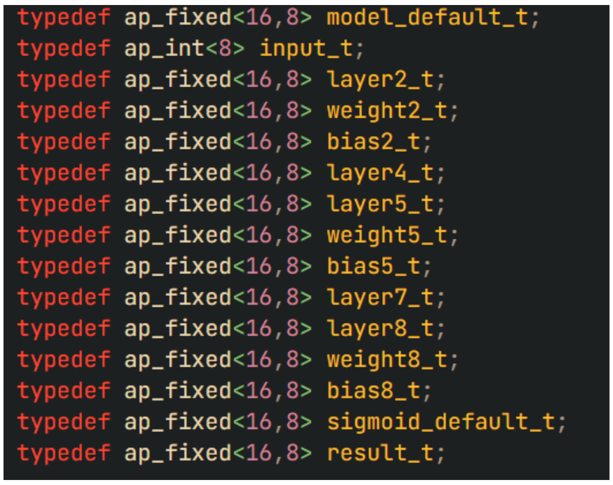
Figure 5.4: After optimization

Since the optimization techniques limit the precision to a certain specified threshold, vary the behaviour of weights and its values, it is essential to observe how these values, their occurrence range and also if they fit under the specified/desired quantization precision. Figure 5.3 and Figure 5.4 portray how the weight values and their precision are distributed before and after tuning and tweaking the network by using the above mentioned optimization techniques. The gray region in the plot represents the selected precision(by user) for representing the values of weights and bias values from each layer respectively. The blue boxes represent the actual values of these weights that occur after training the network. The X axis represents the range of these values and the Y axis consists of the various weights and bias values in the network.

It can be seen that before optimization, the weight and bias value from the first are missing the blue box which represent the occurrence range of the values after training. This means that the actual values of the weights do not lie in the precision specified. This further means that these values are being represented wrongly since not enough bits are provided to accommodate the actual values which might ultimately lead to the end result being inaccurate. It can also be seen that the bias values of the second layer (fc2/b) does not have its entire blue box covered inside the grey region. This means that although the most significant bits (integer or fractional part) of the values are properly represented, the lower most bits (fractional part) are missing. It is always important to ensure that the right part of the box is always inside the grey region meaning that the most significant bits are properly presented. As discussed, previously, it is recommended to consider an extra bit while assigning the precision for the integer part to avoid overflow. The remaining left part of the blue box which is not present in the grey region represents information that is being chopped off due to limiting of the precision. This needs to be properly tuned such that, the precision limiting is just enough to represent the actual values. For example, the values might still be represented correctly even if some lower most bits are skipped since not much information is lost and accuracy drop is negligible. But to attain the best accuracy, it is best to ensure that the assigned bits (grey region) are more than sufficient to represent both MSB and LSB properly taking signed values into account. To maintain uniformity, all the values are given the same precision which is why the grey region is same for different weight or bias value. This can further be fine tuned, having a custom precision for each weight and bias value separately if needed.

After applying optimization techniques, it can be seen that the weights and bias values of all the layers are sufficiently represented where all the grey regions which represent the assigned precision is more than sufficient to accommodate the weight and bias values (blue boxes) both on MSB and LSB respectively. It can also be seen that the values are less scattered (due to regularization) as compared to before optimization where weights and bias values were not present in a selected range. This eases the process of assigning a desired precision and also ensures that the model is more stable when input parameters change. Taking all of the above into account, the network was tuned several times to select the least possible precision without losing accuracy and also maintaining the network stability which resulted in the final design as seen in Figure 5.1.

The finalized network which has been optimized is to be synthesized on the target FPGA. Since High Level Synthesis is being utilized, the network architecture which is written in python is to be converted into equivalent C++ code which can be compiled and synthesized using the HLS tool. For this conversion, the framework hls4ml is utilized which produces a synthesizable C++ code taking into account various specification like target FPGA, desired clock speed and design strategy (latency vs resource minimization). Such a design requires all the values (of weights, bias, input and output) to be represented using fixed point in the resulting HLS C++ code. For this purpose, hls4ml converts the values into data of type *”ap_fixed(X,Y)”* where Y is integer part and X is the total number of bit. This conversion in the C++ code is based on the user specified quantization and the fixed point precision tuning (discussed in previous paragraphs). This can be visualized in the Figure 5.5 which is a snippet of the HLS C++ code which is the same as visualized in the finalized design (Figure 5.1). Here, only the input of the network is represented using the type *”ap_int(X)”* since the input has 8 bit signed integer value as opposed to the weights, bias and output values which are part of the network and might contain floating point values which are restricted and represented using *”ap_fixed(X,Y)”*.



```
typedef ap_fixed<16,8> model_default_t;
typedef ap_int<8> input_t;
typedef ap_fixed<16,8> layer2_t;
typedef ap_fixed<16,8> weight2_t;
typedef ap_fixed<16,8> bias2_t;
typedef ap_fixed<16,8> layer4_t;
typedef ap_fixed<16,8> layer5_t;
typedef ap_fixed<16,8> weight5_t;
typedef ap_fixed<16,8> bias5_t;
typedef ap_fixed<16,8> layer7_t;
typedef ap_fixed<16,8> layer8_t;
typedef ap_fixed<16,8> weight8_t;
typedef ap_fixed<16,8> bias8_t;
typedef ap_fixed<16,8> sigmoid_default_t;
typedef ap_fixed<16,8> result_t;
```

Figure 5.5: Fixed point representation in the HLS C++ code

The activation functions used in the network are Sigmoid and Rectified Linear functions. Below, the forumlae used to calculate the sigmoid and rectifier values can be seen.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5.2}$$

$$Relu(x) = max(0, x) \tag{5.3}$$

Equation 5.2 shows the calculation inside a sigmoid activation function (denoted as σ) and Equation 5.3 shows the calculation inside a Rectified Linear activation function (denoted as $Relu$) and x is the incoming input to the functions. Since latency is the target here, it would be efficient to use pre computed values and store them in the memory as opposed to calculating these equation on run time on the FPGA hardware. For storing the rectified linear activation function values, Look Up tables (LUTs) will suffice as they are simple maximization function. However, BRAMs are used to store the sigmoid function values as these might need larger memory to be stored.

Further, the design of the HLS C++ code generated is mainly focused on achieving the lowest latency since that is a key requirement for this project. Therefore, the design strategy involved in achieving in this requires a parallel approach to handle the data inside the network. The HLS C++ code can be embedded with specific directives to the HLS tool which dictate how the resulting RTL design post High Level Synthesis. To improve the latency, *HLS ARRAY RESHAPE* is used on the input of the the design. This directive essentially allows more data to be accessed in a single clock cycle therefore reducing the overall latency of the design. This is done by converting the input data array temporarily into individual elements which is then recombined to form a single wide register. For example, if the input array has X elements each with bit size of Y, then the resulting register will have X*Y bits. This allows us to avoid storing these large input arrays in block ram and instead use register for storing which therefore increases the overall throughput. This is basically a form of loop unrolling executed by the HLS tool to improve latency on the basis of the directive specified.

Similarly another directive namely *HLS ARRAY PARTITION* is used for the resulting output which partitions an array into individual elements. This would essentially reduce the overall throughput at the cost of increased number of registers.

Using the *HLS INTERFACE* directive, the input of the network and the output arrays of the network are specified as input and output ports of the design respectively. This forms the the top level I/O ports of the network.

As already discussed, having pipelined and unrolled loops or functions are beneficial in reducing the number of clock cycle required. The Initiation Interval (II) is the interval between consecutive iterations in such pipelined functions. Using a directive namely *HLS PIPELINE* in the design, the HLS tool is directed to reduce the initiation interval of the design as much as possible reducing the overall latency. By default the HLS PIPELINE tries to achieve the lowest II while achieving the target clock frequency specified (200MHz in this case). Using the above mentioned directives, the HLS tool can achieve a low latency RTL design.

Chapter 6

Results

The previous chapter discussed the resulting network architecture, the optimizations and tuning involved and the High Level Synthesis based RTL design. This chapter discusses in detail, the results that are obtained based on the designed network architecture. Specifically, the resulting RTL simulation results and also the reports resulting from the High Level synthesis on the FPGA. The analysis focuses on the overall accuracy of the design while assessing latency and the maximum input pattern size achievable. Further, these results are compared with traditional techniques like SAD and Interval Matching.

6.1 Accuracy

Before discussing the results, it is important to understand how design attributes like latency and accuracy are measured and it is also necessary to understand the sample set it is being tested with. As already discussed in section 4.1, a set of 10 misaligned side channel traces were obtained from Riscure. Out of these 10 traces, 7 were used to train the network to detect a particular pattern selected (of variable size) by the user which occurs in each trace at certain point on the trace. So essentially, the network is trained upon two input parameters namely the 7 side channel traces and also the selected reference pattern to be detected. To measure how the network design performs in terms of the accuracy, the remaining 3 misaligned traces from Riscure can be used. These 3 traces each contain the pattern to be detected but since these are misaligned, the location of the pattern might be different for each trace. The designed Neural Network is a binary classifier which has an output between 1 and 0 where any value closest to 1 is deemed as a positive result and values closer to 0 are considered as negatives after rounding these values. Since the location of occurrence of the selected pattern on each trace is already known to the user as can be seen in Table 4.1 of section 4.1, the outcome of the Neural

Network which consists of a set of positive and negative results are compared with the actual occurrence location on the trace to find if there exists any false positives or false negatives. These false results provide a measure of accuracy of the network and hence it qualitative performance.

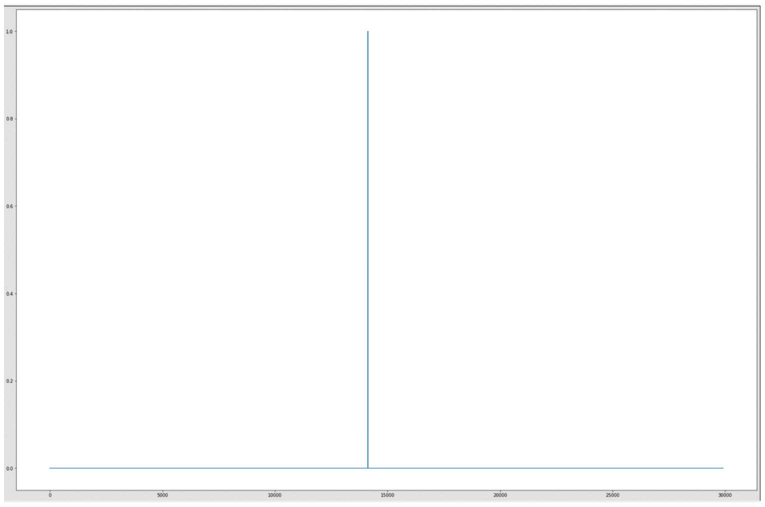


Figure 6.1: Without any Gaussian noise (single detection peak at the expected location i.e true positive)

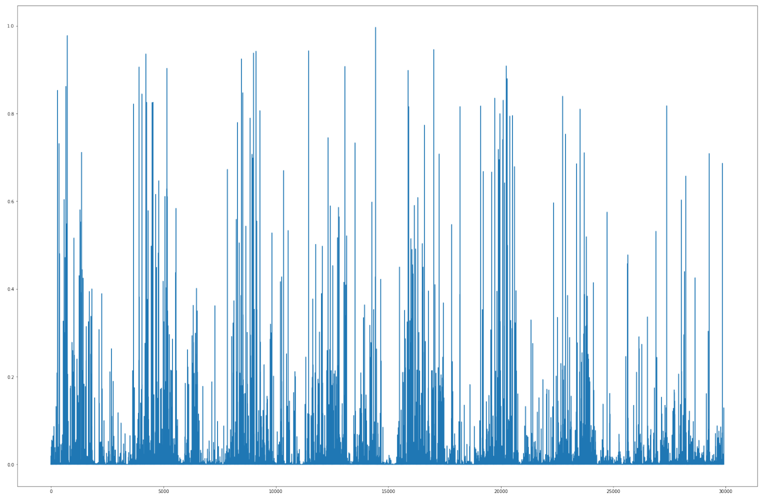


Figure 6.2: With a Gaussian noise factor of 50 (multiple false positive peaks)

The initial testing is done by choosing an input pattern of 84 samples size to be detected and thus the network is trained for the same. It was seen that when tested with the 3 test traces, a 100 percent accurate result is obtained for with only one positive output for each of the trace (i.e the location of the pattern on the trace) with zero false positives and zero false negatives. But since real time signals contain inherent noise, only the testing traces themselves do not provide the complete picture. Therefore, to further assess the network performance under real time signals, Gaussian noise is added to the test traces before testing. Upon subsequent testing with the noisy traces, it is found that if the noise is high enough, it can affect the network performance negatively. It was found that when the trace to be tested had some noise present, the result of the Neural Network sometimes had false positives and false negatives. This can be visualized with Figure 6.1 and Figure 6.2 which are plots portraying the result of the Neural network for a test trace with zero noise and a test trace with a Gaussian noise factor of 50 respectively.

The X axis on the plots correspond to the locations/samples of the trace and the Y axis corresponds to the output value of the Neural Network. In the figure 6.1 (without noise), it can be seen that there is a single large peak occurring in the entire result. This corresponds to the correct pattern detection by the Neural network. On further comparison with the already known location of the trace from Table 4.1, it is observed that the location of the pattern as predicted by the Neural Network is completely accurate without any shift in the location (i.e the Neural Network predicts exact location of the trace). However, the inaccuracies only start to occur under the presence of noise as can be seen in Figure 6.2. Here, it is seen that there are several large peaks occurring in the result in addition to the prominent peak occurring at the expected location. This means that the Neural network when fed with a noisy trace, may consider regions in the trace other than the actual pattern location as a positive result (i.e pattern detected) which means there might be positive results which might be false in reality.

After testing the network with several iterations of noise factors, it was found that the Neural Network would still predict with 100 percent accuracy if the noise factor was negligible in comparison to the upper and lower bounds of amplitudes of the incoming real time signal. However, the network would only begin to predict false positives and false negatives if the noise in the incoming signal was so significant that it would alter the original information making the signal completely different.

To better understand this, one can examine how the pattern to be detected would look like before and after addition of noise. Figure 6.3 shows how the selected reference pattern of 82 samples would look like in the absence of any noise (i.e original). Figure 6.4 shows how the pattern would look like with addition of Gaussian noise factors of 10 and 60 respectively. The X axis in these plots correspond to the samples of the pattern and the Y axis represents the amplitude of the pattern. The added noise factor is relative to the amplitude of the signal. It can be seen (on the left figure) that with a small noise factor, the pattern characteristics and features do not change drastically and therefore such pattern would still be detected accurately by the network. However, information of the trace and pattern completely change when the noise factor is high (right figure) and therefore the network predicts false positives and false negatives. It is hard to quantify and pinpoint how the signal might be affected from various types of noise and therefore the accuracy of the network. Therefore, it is always recommended that pattern with significant features are selected which make make it easier for the network to detect these patterns.

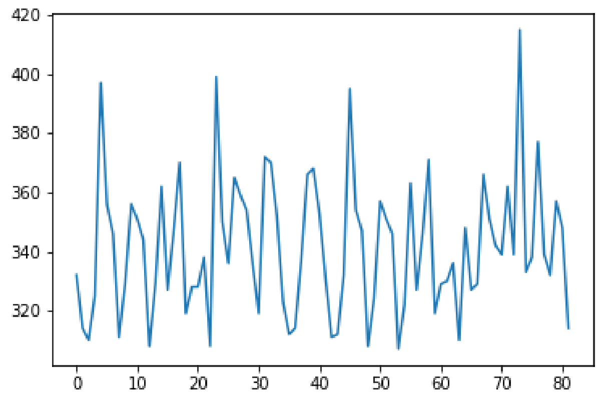


Figure 6.3: Selected pattern (82 sample size) without noise

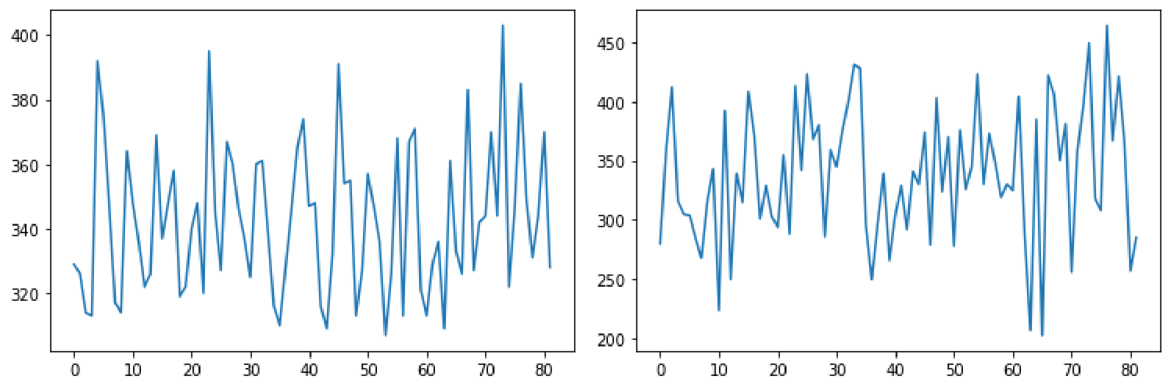


Figure 6.4: Pattern with noise factor of 10 (left) and 60 (right)

6.2 Latency

Further, it was found that there was no loss in accuracy when the Neural network design was optimized and converted into equivalent RTL design for FPGA mapping. This is because in the finalized design architecture, the selected precision after tuning(discussed in chapter 5) is sufficient and no information is lost due to limiting of fixed point precision. Therefore, the simulation result based on the RTL design obtained after High Level Synthesis is same as the result of the network before any optimizations, but the resulting pattern detection occurs after a certain delay which corresponds to the latency of the design on the hardware. Figure 6.5 shows the simulation result of the RTL design after converting Neural Network design using HLS.

As can be seen in the figure, even the RTL result (highlighted layer10 out in the figure 6.5) follows an accurate result with a single significant peak at the expected pattern location. Further, figure 6.6 shows that the peak occurs at adc stream offset (2nd from top in the figure) of 14883. The adc stream offset basically represent the incoming test trace in this case and this means that the peak occurs at 14883rd sample on the test trace. However, the actual location of the pattern in the trace is at 14847th sample. There is difference of 35 samples between the actual occurrence location of the pattern and the resulting location detected by the network design.

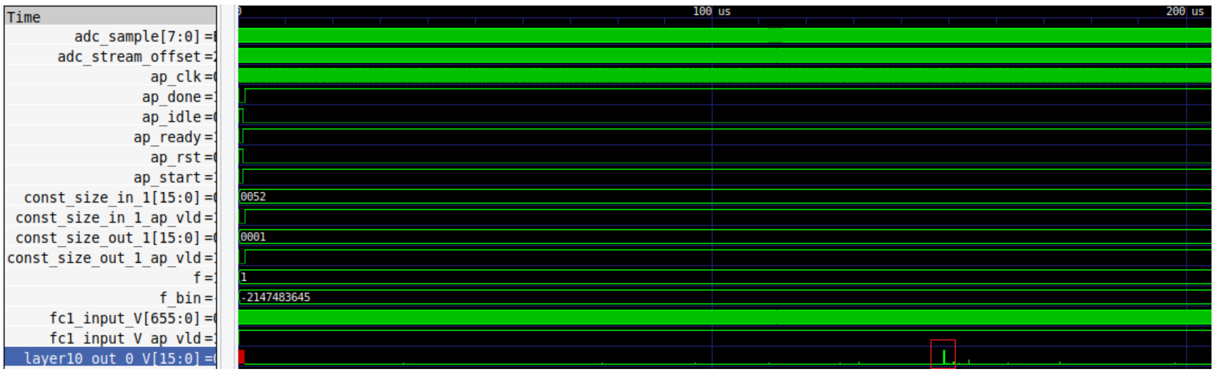


Figure 6.5: RTL simulation result(Peak Detection)

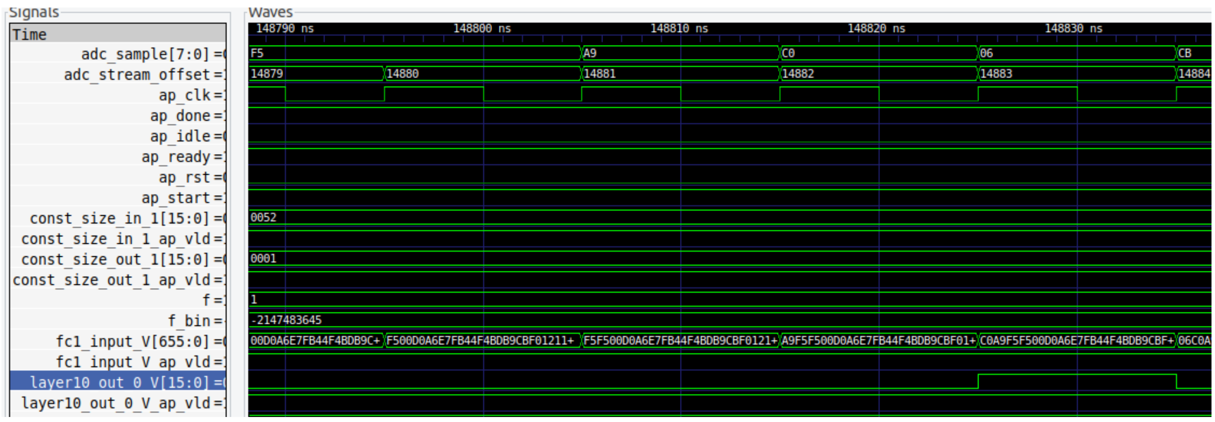
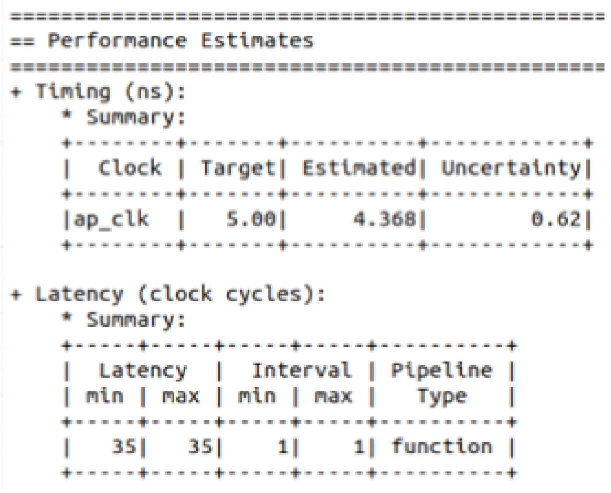


Figure 6.6: Occurence location and clock cycle delay

Upon examining the timing report resulting from the High Level Synthesis as seen in Figure 6.7, it can be observed that the latency of the design corresponds to 35 clock cycles which explains the shift in the result by samples. The initiation interval between two pipelined operations is also minimized and is 1 clock cycle. Further, it can also be seen that the achieved estimated timing of the clock in the design is 4.368 which corresponds to a clock frequency of 228.9 MHZ which is well over the specified target frequency of 200 Mhz assigned during the HLS process. Therefore, the overall latency of the design is 35 x 4.368 ns = 152.88 ns.



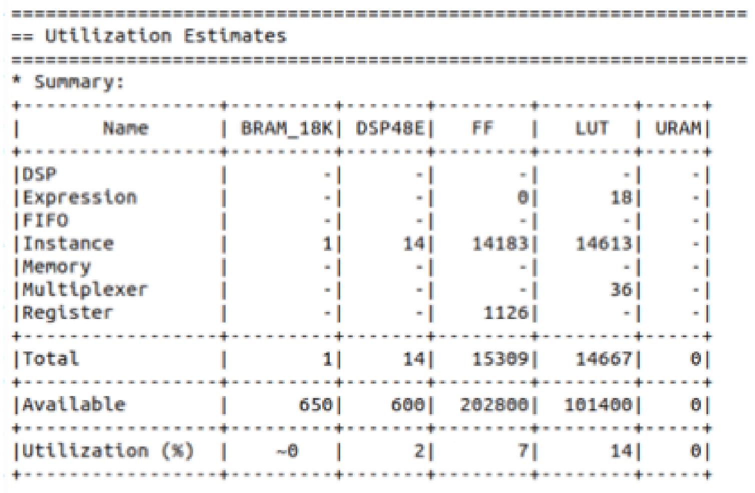
```
================================================
== Performance Estimates
================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |   5.00|     4.368|        0.62|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+----------+
    |  Latency  |  Interval | Pipeline |
    | min | max | min | max |   Type   |
    +-----+-----+-----+-----+----------+
    |   35|   35|    1|    1| function |
    +-----+-----+-----+-----+----------+
```

Figure 6.7: Post Synthesis timing report

6.3 Resource usage

Figure 6.8 portrays the total resource utilization report on the target FPGA after the finalized design is synthesized using Xilinx HLS tool. It can be seen that post fixed point optimizations and tuning that were implemented, the resource usage on the FPGA reduces drastically as compared to that before optimization as discussed previously in Figure 5.2 of chapter 5. It can be seen that the DSP48E usage is improved. This is because these DSP blocks which are responsible for the multiplication operations in each neuron now have fixed point data instead of floating point. This also holds true for the addition operations on the neurons which are done using the logic cells. The Flip flops (FFs), Look Up Tables (LUTs) and register usage corresponds to the way the data is handled inside the network. On the input layer of the network, the incoming data stream which is an array of feature size (in this case 82 which is the pattern size) is transformed (using HLS ARRAY RESHAPE directive) and unrolled into a single register of bit width of size 82 x 8 = 656 which is equivalent to 82 samples of 8 bit each. The next layer which is a dense layer accesses these values from the register and therefore improving read and write speed. Similarly, the consecutive intermediate layers and the output layer use HLS ARRAY PARTITION directive to decompose output array into smaller registers to increase the read and write speed at the cost of increased number of registers. Consequently the input layer array reshaping process also require multiplexing (656 to 1) and demultiplexing accordingly. But these directives which essentially process data by unrolling also increases the number of registers used significantly. Although the report portrays low usage of the various resources with sufficient room to implement a larger design in theory, it is important to keep in mind that due to design strategy (unrolling and pipelining) to achieve low latency, the amount of registers used and multiplexer size might have to be limited based on the requirement of latency (i.e limiting read and write, therefore the overall clock cycles required).



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+--------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |   LUT  | URAM|
+-----------------+---------+-------+--------+--------+-----+
|DSP              |        -|      -|       -|       -|    -|
|Expression       |        -|      -|       0|      18|    -|
|FIFO             |        -|      -|       -|       -|    -|
|Instance         |        1|     14|   14183|   14613|    -|
|Memory           |        -|      -|       -|       -|    -|
|Multiplexer      |        -|      -|       -|      36|    -|
|Register         |        -|      -|    1126|       -|    -|
+-----------------+---------+-------+--------+--------+-----+
|Total            |        1|     14|   15309|   14667|    0|
+-----------------+---------+-------+--------+--------+-----+
|Available        |      650|    600|  202800|  101400|    0|
+-----------------+---------+-------+--------+--------+-----+
|Utilization (%)  |   ~0    |      2|       7|      14|    0|
+-----------------+---------+-------+--------+--------+-----+
```

Figure 6.8: Post Synthesis resource usage report of the final design

6.4 Maximum Pattern Size and limitations

Further, the experiment is repeated with a larger pattern of 154 samples size and it is observed that the resulting design has an increased latency with the increase in sample size. As opposed to 35 clock cycles for the 82 sample based design, it now requires 50 clock cycles for the 154 sample based design. The achieved clock timing is 4.296 ns which corresponds to a clock frequency of 232.77 Mhz which is similar to the previous case. The overall latency is hence 214 nanoseconds. This again is due to the increased read and write operation based on the increased pattern size. The FPGA resource usage reported is still low in terms of the DSP, BRAM, FF and LUT usage, but the input unrolling procedure now requires a 8 x 154 = 1232 bit width register and also the same trend follows for the multiplexer size and number of registers and used subsequently as seen in the previous case. Therefore, it is observed that with the increase in pattern size two main factors are affected namely the latency and register usage for the read or write operations.

When further experimenting with larger pattern sizes, it is seen that the network synthesis starts failing after 310 samples size. Upon detailed inspection and analysis, it was discovered that the reason behind this is due to the limitations from the design strategy and also the specified target clock frequency and timing. Specifically, the number of registers required at this pattern size to handle the data would increase and reach the threshold after which it would not be possible to meet the target frequency specified and also achieve the desired low latency. If larger patterns ($>$310 samples) are to be incorporated, then the design strategy would also have to be varied such that additional memory elements like BRAM would have to be employed which in turn would affect the latency of the design. However, these are not considered here since the scope of this thesis is to focus on latency primarily.

6.5 Flexibility

Previously, various metrics like accuracy, pattern size and other hardware based metrics like latency, clock speed and resource usage were analyzed. However, another important factor to be considered is the flexibility of the overall design and its testing workflow for the particular SCA and FI scenario. The Neural network based design for a SCA and FI pattern detection specifically on an FPGA based device like icWaves is a new approach to improving the already existing designs with traditional techniques like Sum of Absolute differences and Interval matching. The implemented design was possible with the help of modern High Level Synthesis tools and supporting frameworks which are built around optimizing Neural network designs for FPGA inference. The design and testing workflow involved is different in comparison to the traditional techniques which are built directly using RTL. The core principle revolves around training neural network design on a higher level (using software on a computer) and then mapping the resulting network with all its weights and bias values onto the FPGA by leveraging modern HLS tools and surrounding machine learning frameworks. It is seen that the procedure begins by creating a relevant data set using side channel information (set of traces or signals) from the device under test and also selecting a particular arbitrary region on the collected trace which is the pattern to be detected. It is important to note that although the network architecture is fixed (neurons and quantization), the training and synthesis steps need to be executed every time input parameters of the neural network change (i.e dataset created on the basis of collected signals and the selected pattern in the signal) and therefore a new RTL design is created each time the synthesis is done based on the updated weight, bias, input and output values. It is necessary to note the implications this might have on the overall side channel and fault injection testing scenario. On one hand, having an optimized network architecture (fixed quantization,neurons,learning rate) which already has been tested for feasibility on the target FPGA can be beneficial. This is because altering and testing various different input scenarios like pattern characteristics, pattern sizes, noise factors etc becomes easier as these first tested on a higher level for feasibility and possible failures which is followed by the final design mapping if everything works as intended and further FPGA design metrics can be evaluated. On the other hand, it is also important to consider the time required to achieve this workflow (i.e training and synthesis time) on each input parameter variation. It is observed that the network is a small MLP architecture with labelled dataset which can be trained in relatively less time (max of 5-10 minutes) and the synthesis of the network on the FPGA is what consumes the most time (around 30 minutes to an hour). Therefore, the overall workflow achieved here hinders flexibility when considering the need to synthesize the design repeatedly and also the processing times involved. However, there is an added benefit of accessing and modifying the network design at a higher level which can be useful.

6.6 Comparison with traditional techniques

6.6.1 Hardware metrics

Table 6.1 shows how the hardware design of Neural network architecture performs in comparison to the interval matching and the SAD algorithm. Both the Neural Network and SAD algorithms are designed for the icWaves and therefore use a Xilinx Kintex-7 XC7K160T FPGA (162k logic elements). The authors of [7] use an Altera Cyclone IV GX FPGA (150k logic elements) for the interval matching design.

Algorithm	**Neural Network**	**Interval Matching**	**SAD**
FPGA	Kintex 7	Cyclone IV GX	Kintex 7
Latency	430 ns	32 ns	250 ns
Sample rate(Clock freq)	230 Mhz	125 Mhz	200 Mhz
Sample size(in 8 bit)	310	2625	1024

Table 6.1: Comparison of different algorithms

The latency of the Neural Network design depends on the size of the input pattern and as already discussed in the previous chapter, the latency increases linearly with increase in the pattern size. The 430 ns latency of the Neural network design as seen in the table corresponds to the design obtained for the maximum possible pattern length of 310 samples. It can be seen that interval matching has the lowest possible latency and the maximum reference pattern size although it has a lower clock speed. The authors of [7] mention that the originally achieved frequency of the design was 171 Mhz but had to use a lower overall frequency due to the need for down sampling the input signal.

Considering the FPGA Resource usage, the interval matching consumes around 88 percent of logic units on the cyclone FPGA while the SAD consumes around 80 percent on the Kintex 7 FPGA while achieving the results as seen in the table. In comparison, the FPGA design of the Neural Network can only accommodate a maximum of 310 samples but reports a very low resource usage of 15 percent on the Kintex-7 FPGA for the same. However, it is very important to note that the design does not allow inputs of larger samples sizes as is the case in the Interval Matching and SAD algorithms due to design strategy limitations as discussed in section 6.4 which is a pitfall of the design in terms of resource utilization.

6.6.2 Design Flexibility

In terms of design flexibility, both the SAD algorithm and the interval matching algorithm use a separate module on the FPGA called capture module to capture a region or pattern of interest from the incoming real time signals and then store it in the memory which is done before the actual algorithm is executed. Then during the execution of the algorithm, this pattern to be detected are accessed from the memory. In comparison to these algorithms and their flow, the Neural Network design flow already captures this information pre synthesis and the designed Neural network will accept input which is of the size equivalent to the input feature size(in this case the pattern is the feature of the input stream). However, the overall flexibility is hindered as previously discussed in section 6.5 due to the repetition of the entire flow and also the synthesis process based on input changes which is a disadvantage compared to the SAD and Interval Algorithm.

6.6.3 Accuracy

Since both the SAD and interval matching algorithms are signal similarity algorithms their FPGA based RTL design is based on a sliding window concept where the incoming real time signal is compared to a window of size equivalent to the reference pattern size where a barrel shifter is used to send incoming data sample by sample (each of size 8 bit) into the window. The result of the comparison is therefore the mean of results over the entire window which means that it wont be one single peak as seen in the case of the Neural Network design. Rather, the result will be a gradual change over the region of interest (pattern location).

In the SAD algorithm, if the resulting mean sum value is greater than a specified threshold then the pattern is deemed as a match. Here, the accuracy of the design is highly dependent on the threshold value set. A higher threshold might cause the design to miss the pattern causing a false negative whereas having a threshold too low can result in other parts of the signal being falsely recognized as the pattern which results in false positives (this is specifically important in the case of inherent noise). Therefore, one needs to choose an optimum threshold value to successfully detect patterns.

Similarly, even the Interval matching has a similar working where an additional interval offset is specified over the amplitude of the reference pattern (offsetting on positive and negative part of the signal) to be detected. This algorithm therefore has an extra layer of customization available in addition to threshold parameter to properly assess signals to reduce false positives and false negatives. The authors of [7] who have designed the interval matching architecture explain that they aim to obtain a single correct trigger while particularly avoiding or at least minimizing the false positives. They explain how by tuning the threshold and offset values carefully and after balancing them they obtain a 90 percent correct match and 10 percent false negatives and no false positives when tested upon a real time side channel signal of single RSA branch (cyrptographic process running inside the device under attack) with very low noise.

For testing the Neural Network design accuracy, traces with misalignment are used for testing instead of a real time scenario as seen in Interval matching testing ([7]). In the absence of noise, the Neural Network is able to detect each of these pattern occurrence with a 100 percent accuracy with exactly one true positive per trace and zero false positives and false negatives. This accuracy is maintained sometimes even with a low noise present since the network is still able to recognize and detect the features even if slightly distorted. In some iterations of the experiment with small noise factors, a maximum of 3 to 5 false positives and zero false negatives are seen. The network inaccuracies only start to occur as the noise values become significant. Although the Neural Network design reports high accuracy at low noise values, it is only based on a synthetic testing environment as opposed to testing with a real embedded device as seen in the case of Interval Matching. Moreover, the size of the pattern considered in interval matching design are in the range of 1500-2500 samples whereas in the neural network design it is restricted to 310 samples. Having more sample size also means more features and also more repetitiveness to detect.

Chapter 7

Conclusion

This chapter concludes the thesis by answering the research questions and assessing the feasibility of neural networks as a replacement to existing state of the art techniques.

Looking back the primary research question (section 1.1) which concerns the most important aspects like accuracy and latency, the following assessments are made.

- In terms of accuracy, the neural network design provides good performance with zero false positives and false negatives in the absence of noise and and also has the ability to further adapt to small amounts of noise when tested using misaligned traces. But this result is obtained in a synthetic testing environment as opposed to testing with crytpographic operations on an embedded device at real time. Therefore, further testing and experimentation is required to compare the accuracy with the traditional techniques and make an assessment in this regards.
- In terms of latency, both the interval matching and the SAD algorithm are better since the interval matching algorithm is more than 10 times faster while the SAD algorithm has twice the speed.

Further when considering the second research question which concerns the trade offs between various aspects like maximum pattern size, resource utilization, clock frequency and flexibility of the design, the following assessments can be made.

- The neural network design can support upto a maximum pattern size of 310 which is very low in comparison to the interval matching and SAD designs as seen in table 6.1.
- The neural network design has a better clock performance but at the cost of the maximum sample size.

- The neural network design's workflow which requires re-initializing the entire workflow (from network training to synthesis) based on input parameter changes, hinders the flexibility of the design in comparison to the traditional techniques which do not use a High Level approach.

To summarize, the neural network approach for FPGA based SCA and FI testing using HLS tools is novel and provides a new direction in the domain but still falls behind the traditional techniques in certain aspects. However, having Neural Network for detecting real time traces can be beneficial if the network is tuned and tested further. Since the network design and testing is done at a higher level, this allows for further possibilities with various network architectures that can be implemented in the future along with improved testing and a larger dataset.

Bibliography

[1] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Annual international cryptology conference*. Springer, 1999, pp. 388–397.

[2] J.-J. Quisquater and D. Samyde, “Electromagnetic analysis (ema): Measures and counter-measures for smart cards,” in *International Conference on Research in Smart Cards*. Springer, 2001, pp. 200–210.

[3] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[4] J.-S. Coron and I. Kizhvatov, “An efficient method for random delay generation in embedded software,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 156–170.

[5] C. Clavier, J.-S. Coron, and N. Dabbous, “Differential power analysis in the presence of hardware countermeasures,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 252–263.

[6] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, “Practical optical fault injection on secure microcontrollers,” in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011, pp. 91–99.

[7] A. Beckers, J. Balasch, B. Gierlichs, and I. Verbauwhede, “Design and implementation of a waveform-matching based triggering system,” in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2016, pp. 184–198.

[8] *icWaves Datasheet*, Riscure, 9 2011, version 3c.

[9] M. Carbone, V. Conin, M.-A. Cornelie, F. Dassance, G. Dufresne, C. Dumas, E. Prouff, and A. Venelli, “Deep learning to evaluate secure rsa implementations,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 132–161, 2019.

[10] Z. Martinasek, J. Hajny, and L. Malina, “Optimization of power analysis using neural network,” in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2013, pp. 94–107.

[11] Z. Martinasek, O. Zapletal, K. Vrba, and K. Trasy, “Power analysis attack based on the mlp in dpa contest v4,” in *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, 2015, pp. 154–158.

[12] G. Perin, B. Ege, and L. Chmielewski, “Neural network model assessment for side-channel analysis,” *Cryptology ePrint Archive*, 2019.

[13] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, “Fast inference of deep neural networks in fpgas for particle physics,” *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[14] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[15] C. Coelho, A. Kuusela, H. Zhuang, T. Aarrestad, V. Loncar, J. Ngadiuba, M. Pierini, and S. Summers, “Ultra low-latency, low-area inference accelerators using heterogeneous deep quantization with qkeras and hls4ml,” *arXiv preprint arXiv:2006.10159*, p. 108, 2020.

[16] Q. Ducasse, P. Cotret, L. Lagadec, and R. Stewart, “Benchmarking quantized neural networks on fpgas with finn,” *arXiv preprint arXiv:2102.01341*, 2021.

[17] M. Witteman and M. Oostdijk, “Secure application programming in the presence of side channel attacks,” in *RSA conference*, vol. 2008, 2008.

[18] M. M. Altaf, E. H. Ahmad, W. Li, H. Zhang, G. Li, and C. Yuan, “An ultra-high-speed fpga based digital correlation processor,” *IEICE Electronics Express*, pp. 12–20 150 214, 2015.

[19] H. Kanders, T. Mellqvist, M. Garrido, K. Palmkvist, and O. Gustafsson, “A 1 million-point fft on a single fpga,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 10, pp. 3863–3873, 2019.

[20] R. Schmidt, S. Blokzyl, and W. Hardt, “A highly scalable fpga implementation for cross-correlation with up-sampling support,” in *Impedance Spectroscopy*. De Gruyter, 2018, pp. 81–92.

[21] D. Castells-Rufas, S. Marco-Sola, Q. Aguado-Puig, A. Espinosa-Morales, J. C. Moure, L. Alvarez, and M. Moretó, “Opencl-based fpga accelerator for semi-global approximate string matching using diagonal bit-vectors,” in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 174–178.

[22] T. Van Court and M. C. Herbordt, “Families of fpga-based algorithms for approximate string matching,” in *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.* IEEE, 2004, pp. 354–364.

[23] N. I. Rafla and I. Gauba, “A reconfigurable pattern matching hardware implementation using on-chip ram-based fsm,” in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*. IEEE, 2010, pp. 49–52.

[24] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul, “Accelerating dynamic time warping subsequence search with gpus and fpgas,” in *2010 IEEE International Conference on Data Mining*. IEEE, 2010, pp. 1001–1006.

[25] V. Xilinx, “Vivado design suite user guide-high-level synthesis,” 2021.

[26] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, “Integer quantization for deep learning inference: Principles and empirical evaluation,” *arXiv preprint arXiv:2004.09602*, 2020.

[27] F. Chollet, “Introduction to keras for researchers,” https://keras.io/getting_started/intro_to_keras_for_researchers/, 2020.

[28] FastML Team, “fastmachinelearning/hls4ml,” 2021. [Online]. Available: https://github.com/fastmachinelearning/hls4ml

[29] X. Inc., “7 series fpgas data sheet: Overview,” 2020.

[30] Google, “Qkeras,” https://github.com/google/qkeras, 2019.