FPGA Power Side-Channel Attack

Mark Sears and Erick Rodriguez
Email: (mark.sears and erick.narvaezrodriguez)@utdallas.edu

May 2021

1 Introduction

As requirements on hardware get more stringent with the rise of accelerated computing, data centers are one of the preferred methods to offload tasks that would otherwise have required specialized hardware purchases. While FPGAs and other accelerators have become standard and affordable for those who need them, having access to more expensive or larger setups of these devices can be beneficial for individuals and companies so they don't have to purchase their own. This model of computing on shared resources for loads has helped bring the rise of cloud computing, where tasks can be offloaded to the "cloud" and paying for compute as a service. One aspect, however, that is not widely discussed is the safety of this usage of common devices between two parties. Sharing FPGAs on a cloud server opens up the possibility of designing hardware that can monitor the state of the FPGA. This can lead to exposing transactions and data about other users of the board that was not possible before. We will explore how secure information might be exposed in a "shared" FPGA environment with only remote access to the board. We recreate the work done by Zhao and Suh [1], focusing on the RSA algorithm and how power monitoring with Ring Oscillators can lead to deciphering of the key.

2 Attack Model

This side channel attack relies on measuring and analyzing the power consumption of a chip to infer the key of a cryptographic algorithm. This concept is nothing new, but the paper by Zhao [1] proposes that this attack could be carried out remotely, without physical access, on an FPGA. An attacker could program a portion of the FPGA as a Ring Oscillator power monitor (described below in Section 3). Other parts of the FPGA could be programmed by other users, as in a data center setting. These other applications, running cryptographic algorithms, have unique power signatures based on the key. By remotely monitoring the change in voltage level during encryption/decryption, the bits of the key can be inferred by an attacker.

3 Ring Oscillator Implementation Methods

Our primary purpose in this project is to recreate the results of [1] as closely as possible. The major parts are:

1. Ring Oscillator (RO)
   - Used to measure power consumption on the FPGA
2. Power Viruses
   - Used to verify that the RO accurately measures power
3. RSA Algorithm
   - Cryptographic algorithm from which we attempt to steal the key

Each of these parts are described below in detail. Wherever possible, our implementation is compared to the paper [1]. Project files can be found at https://github.com/Zadielerick/fpga-power-side-channel-attack.

We implement this project on the Zybo Z7 development board, using a Zynq7000 FPGA. For compiling designs into bitstreams, we use Xilinx Vivado [2] (2020 version).

3.1 Ring Oscillator

The Ring Oscillator was implemented as close to the paper's description as possible [1]. The principle is simply that an odd number of inverters are arranged in a ring and the output will oscillate at a frequency depending on the gate delays. An AND gate is used to add an enable signal to the RO so that it can be turned on and off.

One problem with creating an RO is that any good synthesis tool will optimize extra inverters out of the design. And so we had to force Vivado to place inverters even when they appear unnecessary. To do this, we used a "LUT1" verilog module to make the inverter for the RO, and declared that we did not want it optimized by using the verilog attribute:

```
(* LOCK_PINS = "ALL" *)
```

Using this statement locked the LUT pins so that the synthesizer would not modify the inverter in any way.

Another challenge occurred when generating the bitstream for the RO's, because Vivado gave the error "LUT Combinational Loop Alert" which was caused by the feedback within the RO. Normal combinational logic should avoid this due to race conditions, but in our case we want it as part of the intended design. To force Vivado to keep the loop, we added to the Xilinx Design Contraints (.XDC) file the following condition:

```
set_property ALLOW_COMBINATORIAL_LOOPS TRUE
    [get_nets design_1_i/RO_counter_*/inst/osc_1/osc_out]
```

For this condition, the "*" indicates that all RO counters should have this property applied. A single RO as implemented is shown in Fig. 1. Compared to the target design, our implementation has an extra AND gate, which is a consequence of using the `LOCK_PINS` attribute on the LUTs. The logic is unchanged, and a single inverter produces an oscillating output. This oscillation feeds into a toggle flip-flop (TFF) binary counter to track how many oscillations occur while the RO is enabled. The counter uses TFFs because the oscillations are orders of magnitude faster than the system clock (GHz instead of MHz). We implemented 16 ROs, and add their counts together to obtain a power monitor signal. As outlined in the Zhao paper, the frequency of these ROs will vary based on the supply voltage. Even small changes to the supply can have a detectable change in frequency. Specifically, the more power that is consumed on the chip, the lower the supply voltage will be dragged down, and this lowers the RO frequency which can then be detected. In Eq. 1, the RO frequency depends on a base frequency f_0, a coefficient k, and the supply voltage at a particular place and time $V(x, y, t)$.

$$f_{RO} \approx k * V(x, y, t) + f_0 \tag{1}$$

3.2 Power Viruses

The next step was to implement a power virus: a module on the FPGA which does nothing except drain power at a predictable rate. Having a power virus allows us to verify the accuracy of the RO by carefully controlling how much power is consumed by the chip at any moment. The power virus consists of a DFF, inverter, and AND gate. An example power virus can be seen in Fig. 2 compared to the design from the Zhao paper. During implementation, the Vivado optimizer merged the gates of the power virus into a single LUT, instead of two separate gates, but the logic remains the same. When enabled, the virus flips the DFF every clock cycle, giving an activity factor of 1. By implementing many thousands of these viruses to fill the FPGA, we can get a wide range of power levels that are easily controlled.

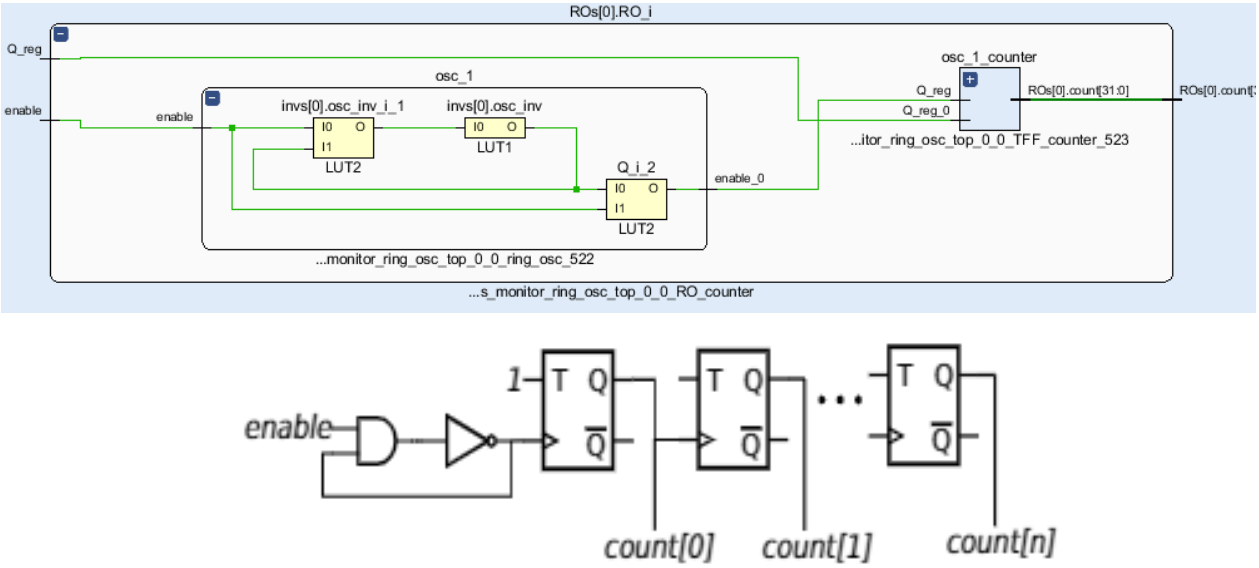

Figure 1: Our RO implementation vs target design

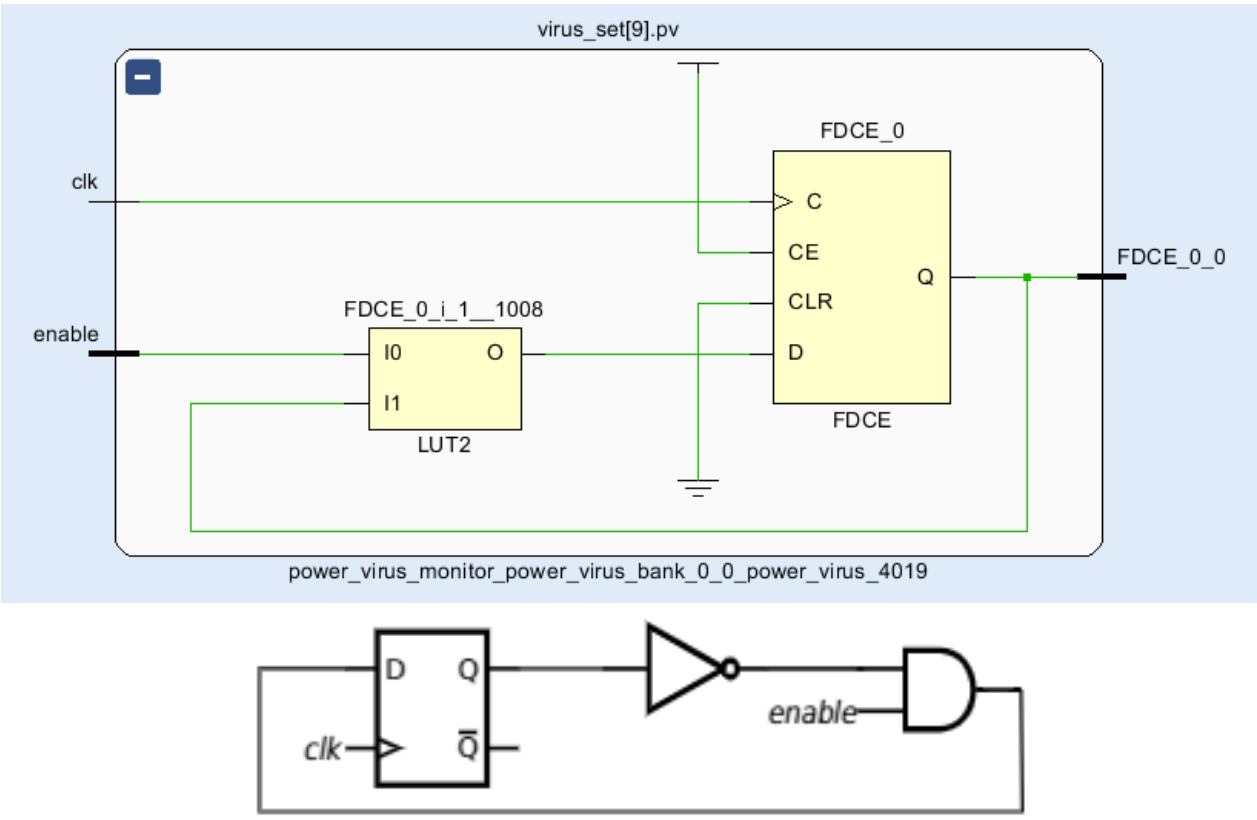

Figure 2: Our power virus implementation vs target design

3.2.1 Difficulties

There were a number of difficulties in the implementation of the power virus. The Vivado compiler interpreted the power virus as unneeded, because no useful logic is performed. After attempting various solutions to circumvent the compiler, the solution we found was to add a dummy output to the power virus, and XOR all the virus outputs together so that Vivado sees them as being useful logic, and will not optimize them out of the design.

3.3 Testing

To demonstrate that the RO power monitor is functioning correctly, we instantiate 16 ROs in verilog. To see into the FPGA signals, we use the Vivado debug modules:

- Integrated Logic Analyzer (ILA)
  - Reads an internal signal through JTAG
  - Generates a waveform of the target signal
  - Uses a significant amount of BRAM
- Virutal Input/Output (VIO)
  - Allows real-time monitoring of FPGA signals
  - Provides an interface for injection of control signals into the design

By using these debug modules, we were able to get results using only the Vivado Hardware Manager. This allowed us to avoid the complications of adding in the Zynq processing system to handle data, and run the design entirely on the FPGA fabric.

With the debug modules in place, we filled the rest of the FPGA space with power viruses. For the Zynq7000, we instantiated 22 sets of 500 power viruses for a total of 11,000 viruses. This utilized 95% of the device LUT capacity. Each of the 22 sets had a separate enable signal, which could be manually turned on or off using the VIO module. We turned on progressively more power viruses, and took measurements of the RO power monitor count using the ILA. A waveform was read from the FPGA, and converted to a csv file for analysis. The total RO counts were recorded and averaged over 13ms intervals, which was the maximum time that could fit on the Zynq BRAMs. The average RO counts can be converted to frequency estimates using Eq. 2:

$$f_{RO} = C_{RO} * \frac{f_{ref}}{C_{ref}} + \epsilon \tag{2}$$

Our reference frequency and reference count remain constant (100MHz and 4096 cycles, respectively). For simplicity, we ignored the ϵ correction since it only makes for slightly more accurate measurements.

After sampling the RO frequency with progressively more power viruses enabled, we obtained the regression shown in Fig. 3. Each data point shows the computed RO frequency with a particular number of power viruses active. We find an R value of 0.9432, which is not nearly as good as the Zhao paper's value of 0.9966. Still, this demonstrates that our RO power monitor is capable of distinguishing power consumption on the FPGA.

Additionally, we used the Vivado system monitor (also known as the XADC) to track the logic supply voltage (vccint) and temperature on the FPGA in real time. Fig. 4 shows a significant difference is observed when there are no power viruses enabled (left) and when all 11,000 viruses are enabled (right). Note that when many power viruses are on, the voltage drop by roughly 1%, which corresponds to a decrease in the RO frequency. This means that the lower the measured RO frequency, the more power is being consumed on the chip at that moment.

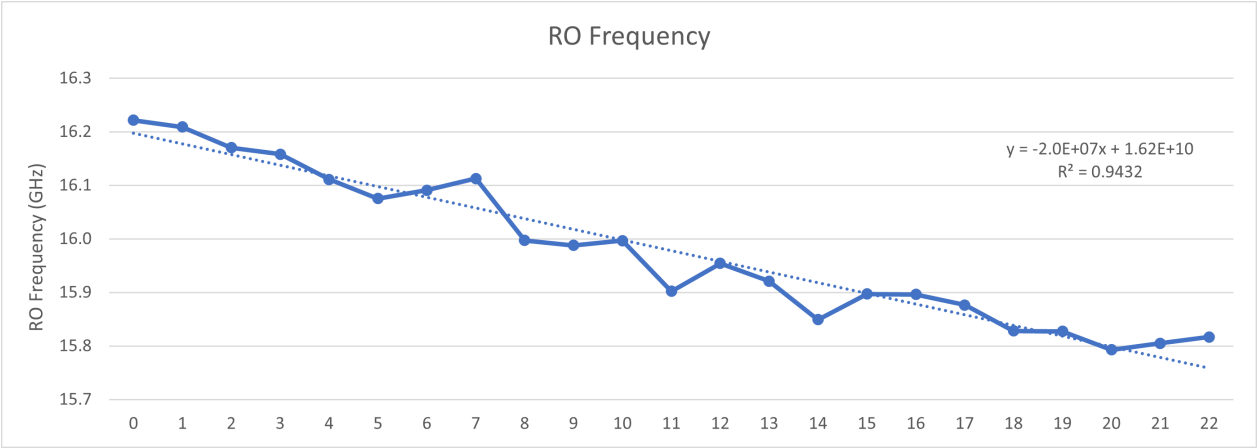

Figure 3: Power Virus Regression

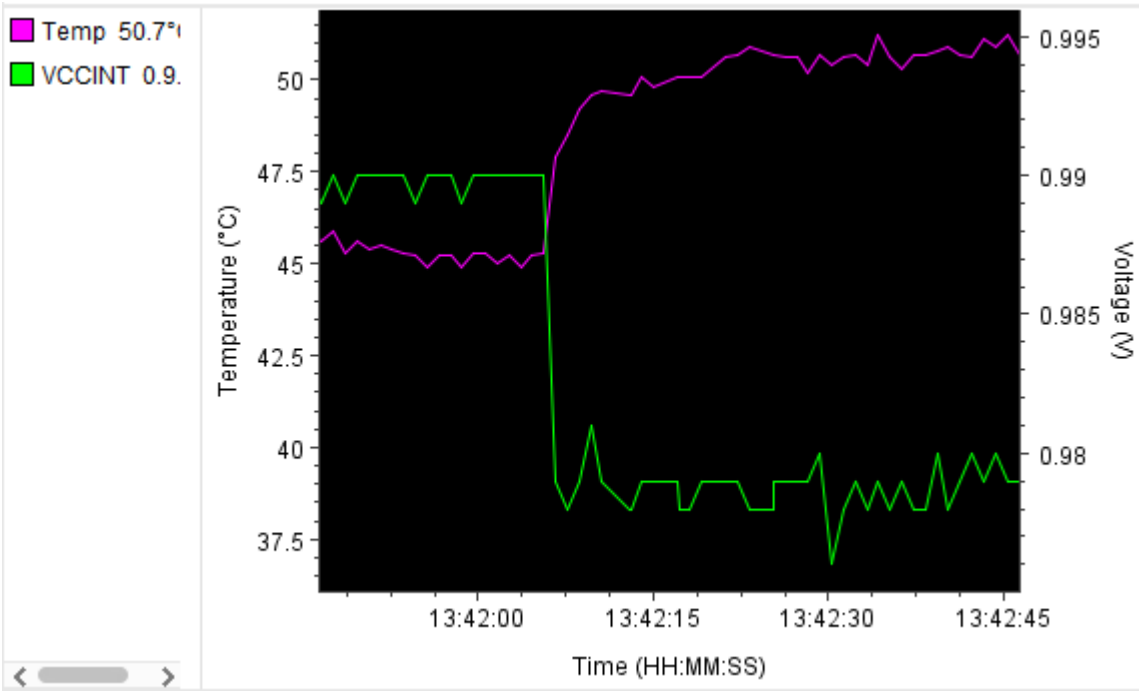

Figure 4: Change in supply voltage due to power viruses

3.4 RSA Algorithm

Next, we implement an RSA algorithm and attempt to steal the key. The RSA decryptor takes ciphertext, the private key and the modulus to decipher and output the plaintext. While the key is supposed to be private and only known to the owner, this side-channel attack can determine what it is by the power-levels observed during the RSA decryptor's execution. The algorithm for decrypting RSA in hardware is called modular-exponentiation. As part of the algorithm, the ciphertext representation needs to be raised to a power equal to the private key, and then have the modulus taken, i.e. $M^d \ mod \ N$. Here, the issue of raising a large number to a power can be tricky and quite expensive for hardware resources.

An algorithm is used where instead of directly raising the ciphertext to a power, an iterative multiplication approach can be used to determine the result directly, without needing to compute the exponentiation. At the same time, because of modulus properties, the value can be kept from growing exponentially by taking the modulus at each step. Below is the psuedocode for this algorithm:

```
mod_exp (M, d , N)
{
    R = 1
    S = M
    for (i = 0 to n-1)
    {
        if(d mod 2 == 1)
            R = R*S mod N
        else
            R = R*1 mod N
        S = S*S mod N
        d = d >> 1
    }
    return R
}
```

The RSA decryptor module is made up of a state machine that iterates through each bit in the exponent. It also contains two multipliers that compute the $R = R * S \ mod \ N$ and $S = S * S \ mod \ N$ at the same time. The multipliers themselves are implemented using a shift-and-add algorithm. The multiplier is iterated through each bit of the exponent (the RSA key). If the bit is a 1, the multiplicand is added to the output. If it is a zero, it is skipped. Each iteration the multiplicand is shifted once to the left to increase its value. Finally, the modulus is taken in order to keep the value within the modulus of N.

The goal of the attack is to measure the power while this algorithm is working through the iterations. When the key bit is equal to one, there will be much switching activity since both multipliers in the design will be working. If the key bit is zero, the first multiplier calculates $R = R * 1 \ mod \ N$, which will not change the value and have much less power consumption. Therefore, if we analyze the power using the ROs, we should see a difference when the key-bits are either 0 or 1.

3.5 Implementing Modular Multiplication

The RSA module requires two modular multiplication modules that run together to compute the two terms, R and S, each iteration of the modular exponentiation. They use a shift-and-add approach to compute the multiplication because it allows to exploit the fact that the modulus must be computed at the end. In order to keep the values within the modulus, each iteration of the shift-and-add the modulus is calculated by subtracting N (the modulus parameter) from the product if necessary.

A block diagram of the modular multiplier design is shown in Fig. 5. Register A holds the multiplicand, which is bit-shifted left each cycle. The A - N operation enforces the modulo N requirement, and ensures that the value of A does not grow extremely large. Register P holds the value of the running product, and is also kept within the modulo range each cycle. At the same time, the multiplier (B) is bit shifted in. For a 1 bit, the value in Register A is added to the value of Register P. For a 0 bit, Register P is unchanged. A Finite State Machine (FSM) issues control signals to ensure correct computation, and finishes after all bits

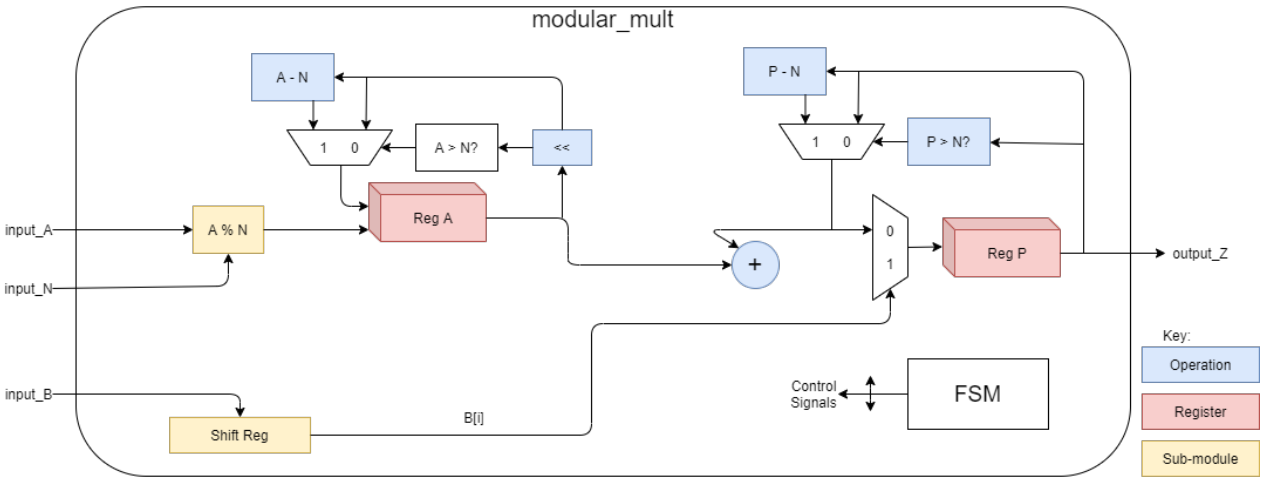

Figure 5: Modular Multiplication block diagram

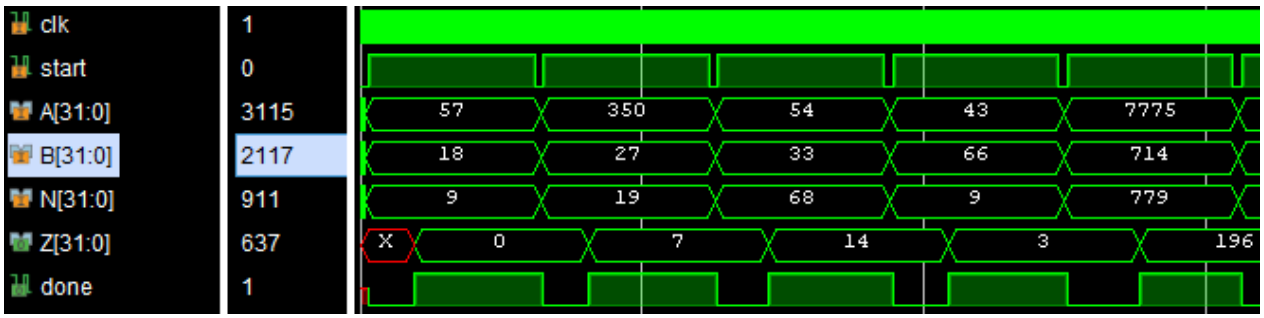

Figure 6: Modular Multiplication waveform

have been shifted in. The result appears on the output Z and is equal to $A * B \ mod \ N$. Notice that if input B was mostly zeroes, then there would be very little switching activity in this module, and hence lower power consumption.

Test benches for the module and sub-modules were created. As seen in Fig. 6, the Verilog implementation was able to correctly calculate the modular product for a few test cases. For example, $43 * 66 \ mod \ 9 = 3$. These were further tested by their use in the modular exponentiation module.

3.6 Implementing Modular Exponentiation

This module will compute the decryption of the ciphertext in the RSA algorithm. This module implements the psuedo code presented earlier to compute $Z = M^d \ mod \ N$.

Fig. 7 shows a block diagram of the modular exponentiation design. The S register is initially set to the M input (the base) and then is updated each cycle using a modular multiplication sub-module which computes $next\_S = S * S \ mod \ N$. Similar to the Modular Multiplication, the exponent "d" (which is the RSA key) is fed in one bit at a time using a shift register. A second modular multiplication sub-module is used to compute $R * P \ mod \ N$ where P is either the S register (when the d bit is 1) or a value of 0x01 (when the d bit is 0). The result is held in the R register. A FSM controls the flow of the computation, and finishes when the shift register is empty. The output Z is read directly from the R register. In this way, modular exponentiation is implemented as per the algorithm.

In Fig. 8, a verification waveform demonstrates the functionality. It can be seen that when the done signal is high, the output on the result line is correct. For example, $8^7 \ mod \ 13 = 5$.

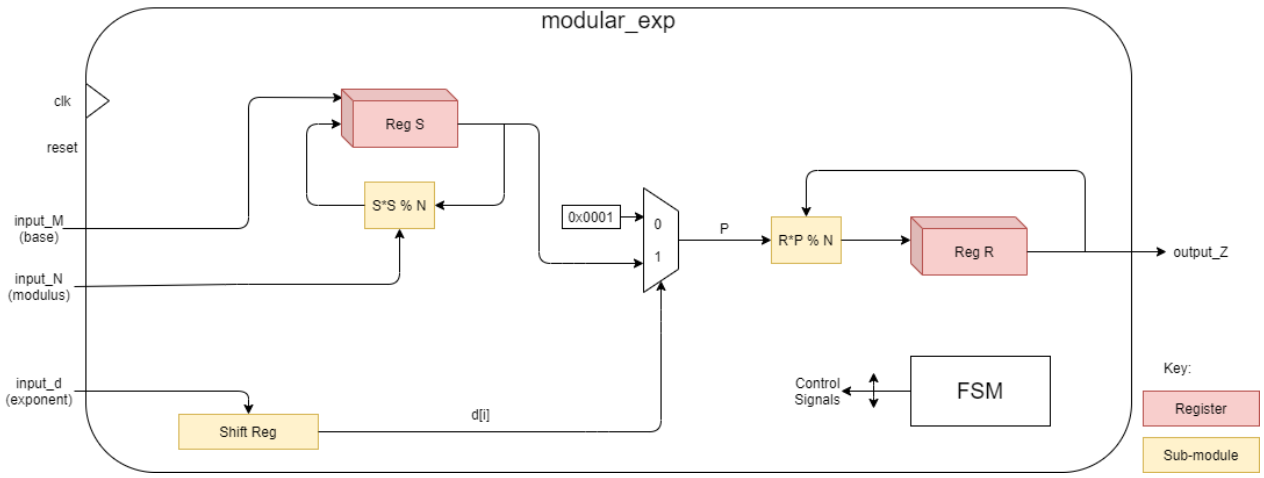

Figure 7: Modular Exponentiation block diagram

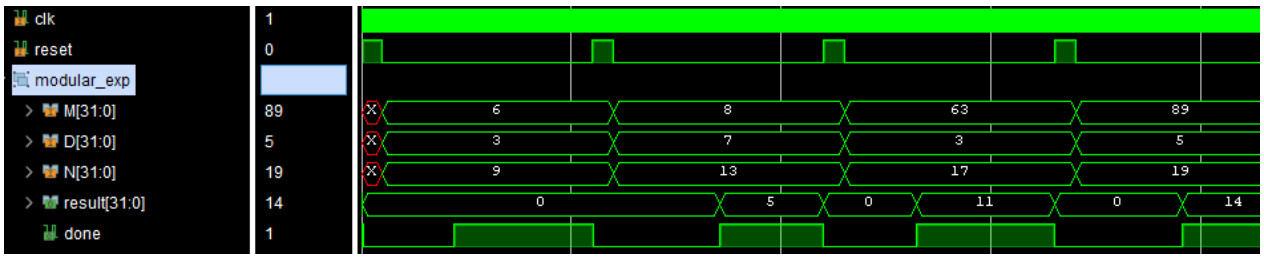

Figure 8: Modular Exponentiation waveform

3.7 Improvements

Several improvements could be made to our implementation with more time and effort.

- The placement of the RO power monitor can have different effects on the accuracy of the attack. Placing the ROs manually near the module to be analyzed is optimal, however, our placement was wherever the tools decided rather than constrained.
- We also tried using the Integrated Logic Analyzer (ILA) IP from Xilinx to measure the ROs power monitor output rather than going through the on-chip CPU as is usually the case. Though this helped keep the complexity of the project low, transferring the data to the ARM core allows more flexibility in data acquisition. The ILA had some limitations as it uses mostly BRAMs and quickly outgrows the size of the FPGA without proper parameters.
- The RSA decryptor design worked correctly only for relatively small inputs. For larger inputs, the output values were incorrect. This was due to timing violations in our design which would make the module unusable in the real world. Further debug would have helped us converge on a more stable design. Since only small inputs were used, this may have made detecting the power differences more difficult, because the computations complete much quicker for small inputs.

4 Results

The finalized Vivado block diagram we used is shown in Fig. 9. The important components are:

- 16 ROs block
- RSA modular exponentiation block
- VIO block for controlling execution
- ILA block for reading RO output counts

With the bitstream programmed on the FPGA board, we took samples of the power monitor as the RSA module ran with test vectors. We collected the samples and plotted them to extract the private key input to the RSA module as seen in 10. Here, the averages of the RO signal was plotted for 16-bits of data, with the average drawn as a threshold line to determine if the key-bit should have been a zero or a one.

Unfortunately, the output data was not representative of the key-bits we were hoping to extract, the noise in the signal as well as the operation of the RSA module internally could have contributed to the unexpected measurements. Also, the size of the RSA module was a constraint. Our board limited the maximum size we could achieve before running out of resources to be around a RSA WIDTH of 380 bits. The goal was to run a module that handled 1024-bit RSA decryption, which would have increased the duration of the algorithm and given better samples with more significant differences in activity.

5 Countermeasures

There are two categories of countermeasures that make sense in the threat model considered in this experiment. The first are countermeasures that have been proposed against power side-channel attacks. Many methods have been proposed before such as randomizing intermediate values so that that the power-analysis of a particular algorithm won't be consistent. Inserting random noise is also another . Though these can make it more difficult to analyze the power to gather meaningful information, they come with performance and energy overheads.

The second category of countermeasures is preventing the attacker from making circuits for power monitoring. When using shared FPGA resources, an admin could analyze the netlist for power analysis designs and block these. These attacks usually rely on combinational loops, and blocking the use of these could also prevent an attacker from programming the FPGA with malicious hardware. The downside is there are also

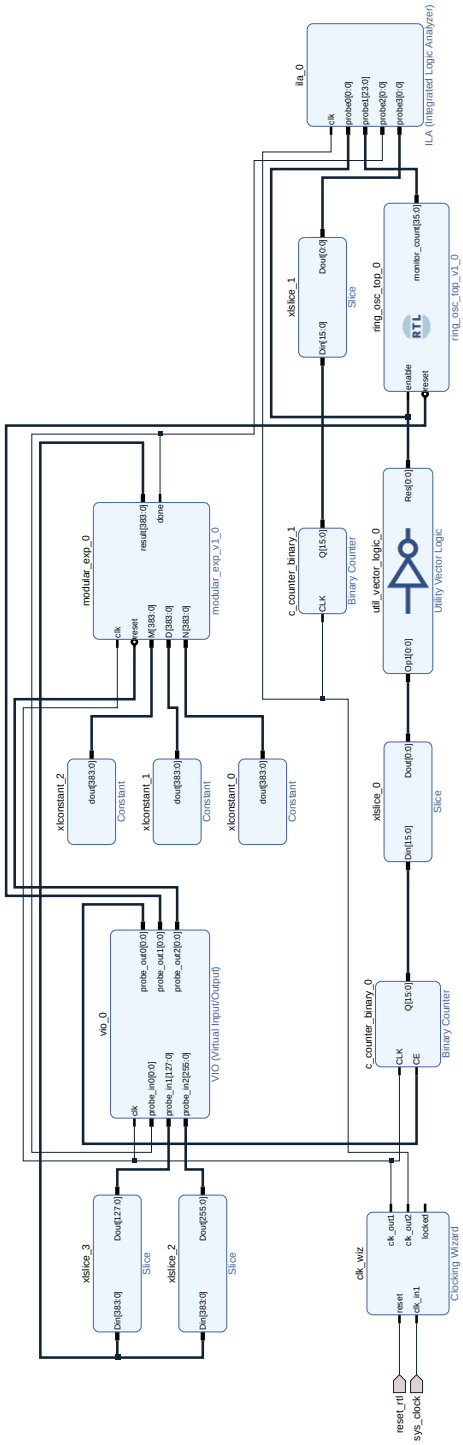

Figure 9: RSA power monitor block diagram

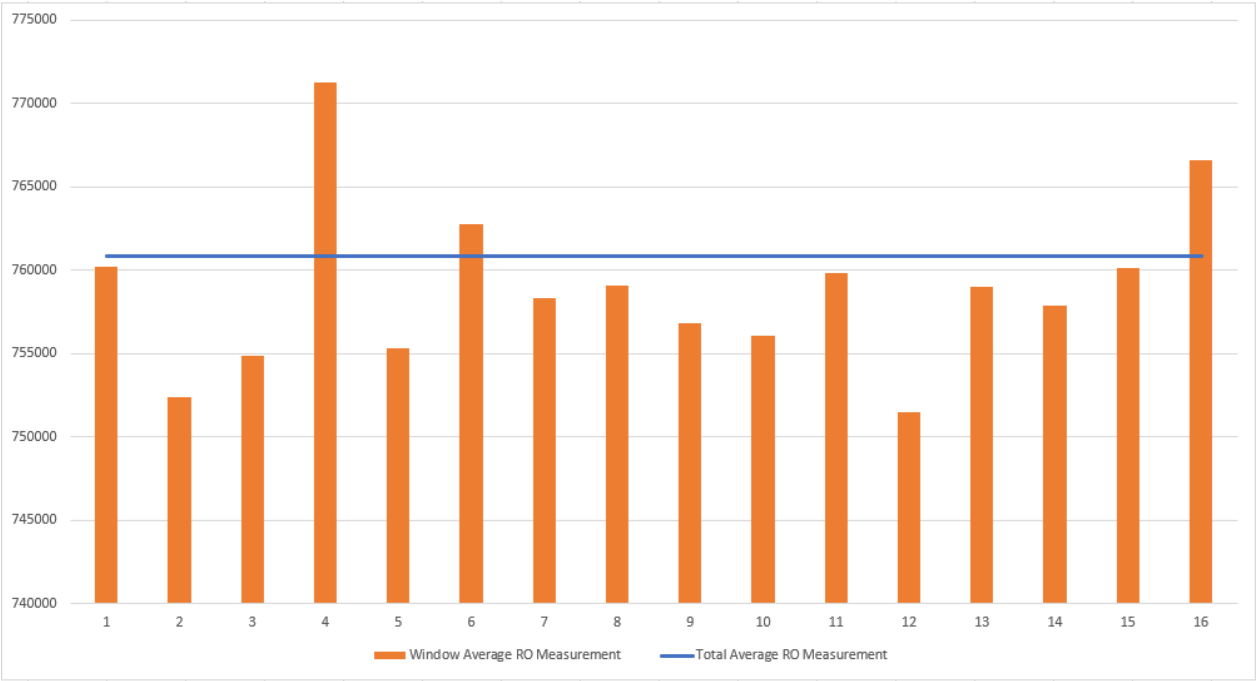

Figure 10: RO Power Monitor Samples' Average

use-cases for these loops that are valid and not meant for attacks, and blocking them limits the ability of the designer from implementing their potential design. The use of an admin can also be too time consuming for every design that is pushed to the board.

Current countermeasures are not sufficient to protect against the side-channel attacks without affecting the design, and requires more research.

6 Conclusion

This project finds that there are vulnerabilities in FPGAs when used as shared resources. Prior work has assumed there are vulnerabilities only when physical access to a device is granted, however, the vulnerability still exists with remote access. A simple RO-based power monitor is capable of detecting fluctuations in the power usage and this information can be used to determine critical values like private keys.

7 Contributions

7.0.1 Mark Sears

- Ring Oscillator Design for Power Monitors
- Power Virus Implementation
- RO Testing with Power Virus
- RSA Verilog Design
- RSA Design Testing on FPGA
- Report

7.0.2 Erick Narvaez

- RSA HLS Design (unused)
- RO Verification
- RSA Design Testing on FPGA
- Presentation
- Report

References

[1] M. Zhao and G. E. Suh, "FPGA-based remote power side-channel attacks," vol. 2018-May, pp. 229–244, Institute of Electrical and Electronics Engineers Inc., 7 2018.

[2] "Xilinx Vivado Design Suite." https://www.xilinx.com/products/design-tools/vivado.htm.