The Design of a Ring Oscillator and Serial Communication System to Complete an FPGA-based PUF

Quillen Flanigan

Conjure-2

8/7/2023

Help Received

- Jim Sloan helped us understand the TPM server/client structure and the CppLinuxSerial repository dependencies
- Cole Rowell and Gianna Bugedi helped us understand the existing functions within CryptRand.c that helped us understand the serial C code

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Abstract

The growing complexity of modern technological advancements has further increased the abstraction between the hardware of the technology and the user. The expansion of technology, and its constant evolution, has forced our society to become dependent on the security of our device's hardware. We must become familiar with hardware security features and tools, along with the dangers with hardware security that are present in modern cyber warfare. To reinforce our understanding of hardware security, we were tasked with designing a ring oscillator using the VHDL programming language to complete an FPGA-based PUF, along with measuring the frequency and gate delay of the ring oscillator. We then were asked to program a Nexys A7 circuit board with our ring oscillator design, and interface this hardware design with a serial port communication protocol. These three main steps combine to build an authentication system for the hardware of our circuit board, allowing us to verify the integrity of our circuit from any potential threats. We were able to successfully design a three-inverter ring oscillator and program our device's FPGA, along with building the necessary code to facilitate serial communication between a TPM server and client. However, we were unable to measure the frequency and gate delay of our ring oscillator. Despite this incomplete solution, we were still able to use our FPGA-based PUF to verify the hardware security of our system through serial communication, making this a valuable tool to use for hardware security in the future.

1. Introduction

Our society has been constantly building new and more complex computing systems ever since we began designing computers nearly a century ago. This increased complexity has abstracted us farther and farther from the inner workings of computers. We now rely on blind trust in the security of our computer's hardware, with little awareness of the operations our computer's actually take to produce our desired results. This reliance on safe, effective hardware can cause severe issues and vulnerabilities within our systems, as modern operators are often ignorant to the threats and dangers to our machines. In 2021, a reported 63% of computer development companies reported at least one hardware security breach [1], depicting an obvious threat to the systems we rely on. These hardware threats show an inherently greater damage compared to software or cyberspace attacks, as the physical nature of hardware attacks make them both very difficult to detect and resolve. Given these dangers, we must be able to detect any hardware anomalies and protect against any potential dangers.

For this challenge problem, we were tasked with designing a ring oscillator within a VHDL program and interfacing this program into a Field-Programmable Gate Array (FPGA) Physical Unclonable Function (PUF) on a provided circuit board. We then were tasked with using this programmed circuit board and a provided C program to create a serial communication protocol over a client-server relationship. Lastly, we were asked to measure the frequency of our ring oscillator and determine the gate delay of one of its individual inverters. These three main tasks will combine to form an authentication system we can use to verify the security of our hardware. By using FPGA-based PUF's and being able to send our programmed circuit pairs of messages and responses, we can analyze the normal circuit behavior and use it to compare to any anomalies that occur in the hardware, which could alert operators to a possible hardware security breach.

1.1. Assumptions and Limitations

While constructing our solution for this challenge problem, we had to address some assumptions we made. Firstly, we are assuming the hardware we are testing on is safe, effective, and working properly. Many hardware security attacks take place in the manufacturing stage of creating hardware, so we have to assume our circuit board has not been breached and will run correctly to give us valid results.

Secondly, we had to assume the serial communication between our server, client and the circuit board was safe from any security breaches and was returning correct values. We were able to establish stable Challenge-Response Pairs (CRPs), but we did assume these would always be the correct values based on the challenges sent over our serial connection. This assumption could limit the accuracy of our results as we were not able to verify the correctness of our response values and could limit our ability to detect a pre-existing hardware anomaly.

2. Background

Most cyber attacks will fall under the “software” characterization, as it is easy to access a machine through a virtual vulnerability or exploit a machine through code. Hardware attacks are much more difficult to implement as it requires physical access to the system, yet they are also much more difficult to discover and resolve due to their low-level complexity and ability to hide from anti-malware detection software. These attacks, such as hardware trojans, are often implemented when the hardware itself is being constructed, such as in a chip manufacturing facility. Once the user gets their manufactured system, it has already been compromised, but most users are often so naive to the low-level nature of their systems that this vulnerability can often go undetected. However, to counter these dangerous threats, operators can use a variety of tools and hardware techniques to help find the threatening hardware anomaly and rectify it.

2.1. VHDL and Vivado

Due to the low level nature of working with hardware, operators must use a specific language that interfaces programs with the circuit board itself. To accomplish these tasks, operators can use the Very High Speed Integrated Circuit Hardware Description Language (VHDL). This language allows users to customize how low they want to get with respect to logic gates, signals, input and output ports, and provides high speed interactions with programmable circuit boards. To run these VHDL programs, users often need a development environment that allows for simulating and analyzing the VHDL code before it is implemented on the hardware. Vivado is a common environment that allows users to run simulations, create visualizations of their VHDL programs, and analyze the performance of their scripts.

2.2. FPGA

Users often want to use VHDL programs to give specific directions to a circuit. To accomplish this, Field Programmable Gate Arrays (FPGA's) allow operators to program a circuit with instructions to complete a specific task, and are able to be re-programmed at any time. This allows flexibility for the user to use their circuit board for several different tasks while providing high speed operations. These FPGAs are constructed out of groups of logic gates, such as AND, NOT and XOR, and configurable logic blocks (CLBs) that allow the user to give instructions to the circuit board.

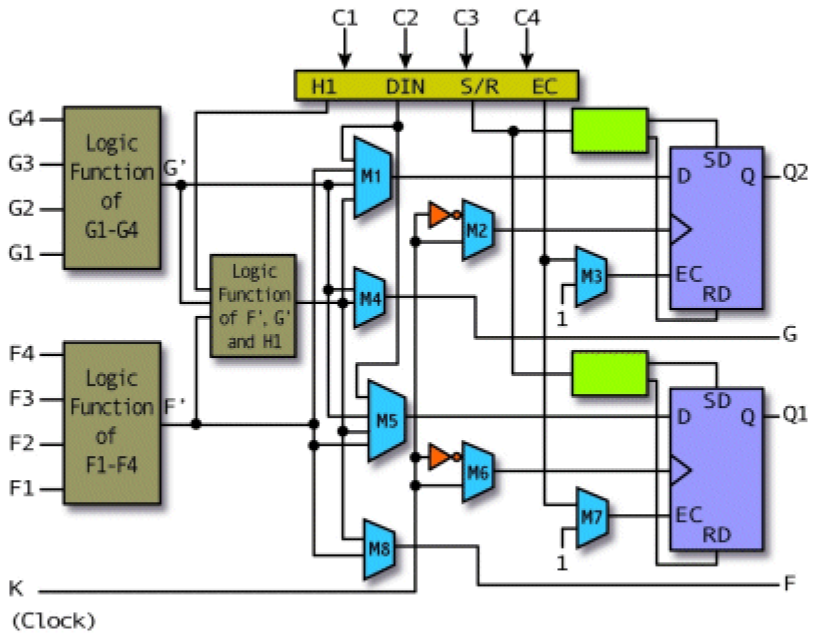


Figure 1. Visualization of an FPGA [2]

Fig. 1 depicts an example of an FPGA, as we see the groups of logic functions together on the left, feeding into groups of multiplexores, and then their output serving as the input into D-flip flops that create the final output of the circuit. The groups of logic functions are the programmable aspect of these chips. These FPGAs also give users the benefit of limited, specific execution, and limit the ability of an adversary to use the FPGA chip to execute their own instructions. FPGAs only execute the instructions they are programmed with, as opposed to a more popular CPU which has unlimited states of execution. While the process of programming, debugging and reprogramming FPGAs for new tasks can be timely, they require physical access to exploit which provides heightened hardware security [3].

2.3. Physical Unclonable Functions

It is easy to forget the complexity of the hardware and inner workings of our systems while we use vast abstractions to simplify the use of our machines. Given this complexity, every computer, circuit board and chip is unique in terms of its physical build and design. Each piece of metal and silicon has variations that make each system unique, meaning the time it takes to process data and instructions is also unique. Hardware security engineers take advantage of these slight variations by using Physical Unclonable Functions (PUFs). Although the variations within integrated circuits (ICs) are unique to every IC and machine they are implemented on, these outcomes are also stable and repeatable once their inputs and outputs are known. PUFs use these repeatable, yet unique, variations as a strategy within hardware security. PUFs are designed to map any input into unique outputs using the individual variation in frequency, voltage and delays, forming the basics of a private key encryption scheme for each IC [4].

2.4. Ring Oscillators

Ring oscillators (RO) are used as a tool to measure the inherent variation within a machine, often with respect to frequency, gate delay and voltage. Ring oscillators are designed as a series of logic gates, often made up of NOT gates to oscillate a given input. This oscillation allows users to measure the frequency of the signal within the IC, using this data to verify the return of a CRP or PUF output.

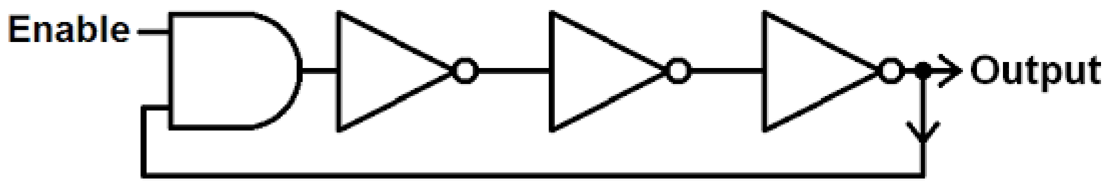


Figure 2. Visualization of a three-inverter ring oscillator [5]

As referenced in Fig. 2, ring oscillators consist of a chain of inverters and often begin with an AND gate to combine the input of a standard Enable signal. The ROs continue their oscillation and allow operators to measure the internal metrics of the specific IC, allowing them to confirm the accuracy of any CRPs and notice any hardware anomalies.

2.5. Serial Port Communication

During the design of ICs and other systems, operators focus on the basic goal of connecting ports and interlinking circuits together so they can communicate and produce an output. For this interaction to take place, the devices must agree on a communication protocol, of which most are categorized as either parallel or serial. Serial ports communicate one bit at a time, and often lack in speed but make up for it in their limited space allocation. Serial communication over ports allow operators to send instructions from a program to a FPGA or IC with simple and efficient implementation [6].

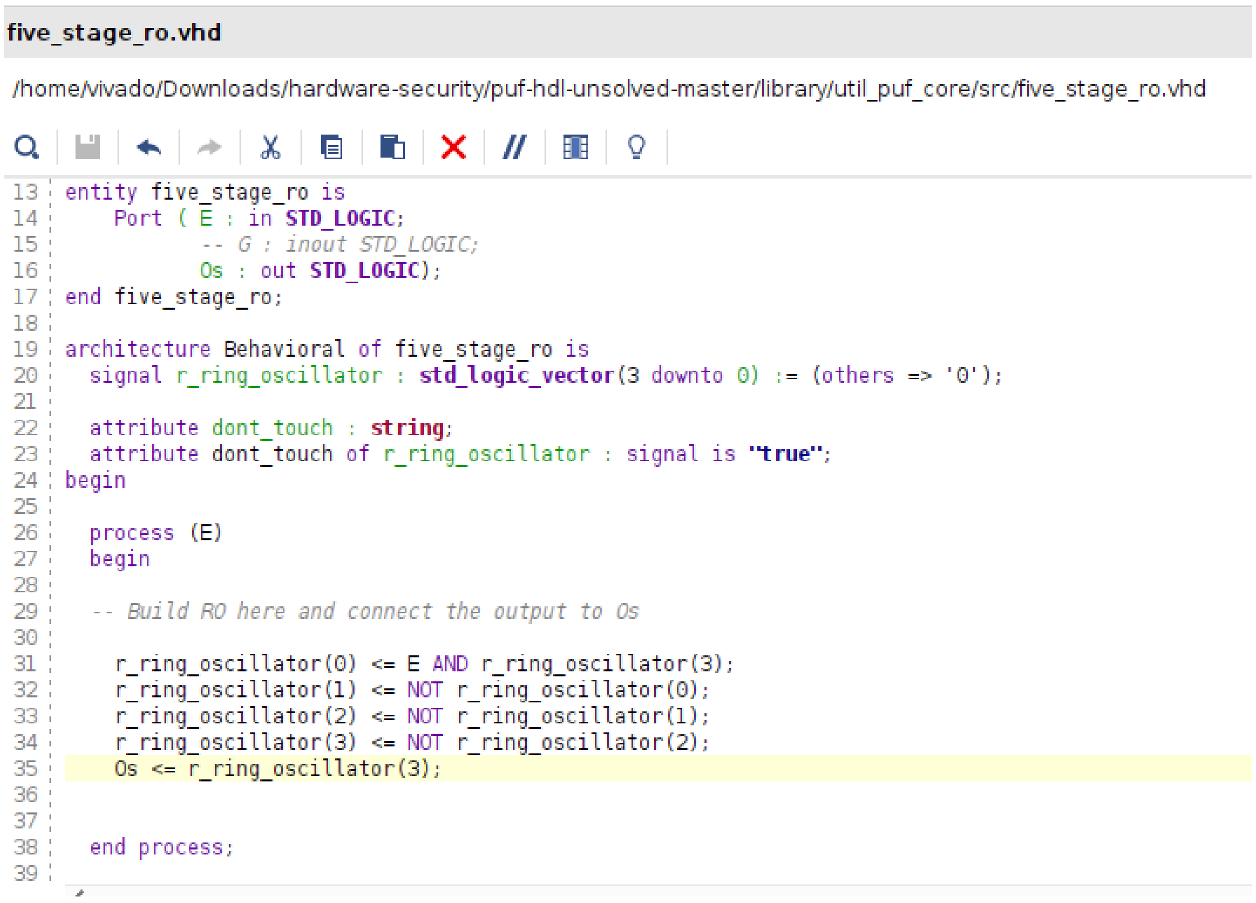
2.5.1 SoftTPM

One of the most common tools used is the SoftTPM software, using a server-client relationship to facilitate the serial port communication needed for the communication from the code to the IC. The server can be run and used to execute programs for the IC and a number of clients can execute commands to call functions within the SoftTPM server and receive responses from the server.

3. Methods

To begin implementing our solution we first had to install a few libraries and download all the necessary dependencies needed for SoftTPM and Vivado to function properly. We began with pulling the GitHub repository “CppLinuxSerial” and following the instructions within its README.md file to complete the necessary setup needed for the serial communication. We also ran the command “sudo apt install <import_name>” for the necessary installations, including arduino, the libssl-dev library, and the moserial serial communication tool. These libraries provided us with the needed functionality to compile each necessary source and header file for the C programs and TPM client-server files to facilitate the serial port communication.

After installing all the necessary dependencies, we began by designing our ring oscillator that would be integrated into the FPGA-based PUF that was provided for us. We were given an existing VHDL program and tasked with using the existing ports and signals to design the ring oscillator logic. The program already contained two ports, an Enable input port and an output port labeled “Os”. There was also a declared signal named “r_ring_oscillator”, declared as a vector with space for four values (3 downto 0). These three variables would be the only things we needed to complete our ring oscillator. The challenge of this portion was using each variable properly to simulate the oscillation of an electric signal, as we had to use the r_ring_oscillator vector to take the place of our Os signal because VHDL does not allow outputs to serve as inputs. As we can see in Fig. 2, the final output of the wire in our RO will serve as the initial input in the next iteration of the oscillator. We used the signal vector to perform this “wrap-around” functionality.



five_stage_ro.vhd

/home/vivado/Downloads/hardware-security/puf-hdl-unsolved-master/library/util_puf_core/src/five_stage_ro.vhd

```vhdl
entity five_stage_ro is
    Port ( E : in STD_LOGIC;
           -- G : inout STD_LOGIC;
           Os : out STD_LOGIC);
end five_stage_ro;

architecture Behavioral of five_stage_ro is
  signal r_ring_oscillator : std_logic_vector(3 downto 0) := (others => '0');

  attribute dont_touch : string;
  attribute dont_touch of r_ring_oscillator : signal is "true";
begin

  process (E)
  begin

  -- Build RO here and connect the output to Os

    r_ring_oscillator(0) <= E AND r_ring_oscillator(3);
    r_ring_oscillator(1) <= NOT r_ring_oscillator(0);
    r_ring_oscillator(2) <= NOT r_ring_oscillator(1);
    r_ring_oscillator(3) <= NOT r_ring_oscillator(2);
    Os <= r_ring_oscillator(3);


  end process;

```

Figure 3. Image of the main ring oscillator code

Fig. 3 references the main syntax used to create the ring oscillator. We began by declaring the first value of our vector to be the output of our Enable input combined with the last value in our vector, with both signals serving as the input to an AND gate. This AND gate allows us to use two inputs, with the Enable pin and our ring_oscillator signal, and only receive a single output which will serve as the electric signal through the rest of our inverters. This AND gate also allows the output of this first statement to always be dependent on our vector signal, as the Enable signal will always have a value of one. The AND gate logic returns a one if both inputs are one, and a zero in every other case. Due to the Enable value always being one, the output of the AND gate will always be the existing signal of the last value in the vector. If r_ring_oscillator(3) is one, r_ring_oscillator(0) will also be one, and vice versa for the value of zero. This allows us to have a completely connected wire signal with the last value looping back to initialize a new iteration, while having the stability that the AND gate provides to the circuit, as opposed to just setting the first vector value equal to the last.

After this AND gate, we designed the rest of the electronic signal to consist of NOT gates, serving to invert each signal and provide the oscillation we need. The repeated oscillation allows us to measure the frequency, gate delay, voltage and other metrics of our hardware to determine any anomalies within the machine. This process of signal oscillation is depicted below in Fig. 4. Our Enable pin is a constant value of one and the last value serves as the input to the next iteration. The purple-colored values represent the current value of the signal and the green-colored values represent the bit of the previous and next iterations. We can see the last value recorded as the value one, so the input into our AND gate is two ones. Thus, the first signal, or r_ring_oscillator(0) is a one. The inverter then flips this signal to a zero, and this process continues until we reach the end value which will be the opposite of the r_ring_oscillator(0) value. As the first value in our signal vector is different from the last signal vector value, the output of the wire is ambiguous and is forced to repeat, creating this oscillation of our signal. Importantly, the ring oscillator contains an odd number of inverters to ensure the signal is constantly oscillating. If there were an even number of NOT gates, the signal would be the same at the last value as it was in the first value, and the circuit would stop executing.

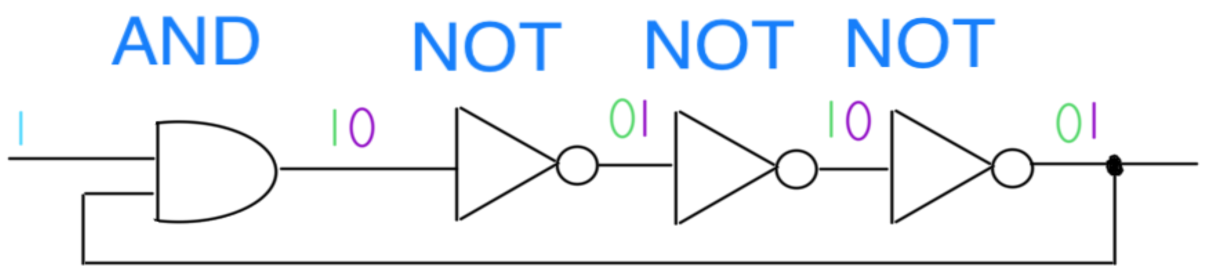


Figure 4. Visualization of the oscillating signals through our ring oscillator

After completing the logic of our ring oscillator code and piping it into the Os output in, we then ran the simulations and synthesis functions provided by Vivado to test the accuracy of our RO. By using these simulations, we were able to generate schematics that would help us verify the correct setup of our ring oscillator. Fig. 5 represents this generated schematic representation, displaying the AND gate taking in our two inputs and the flow of this output into each inverting NOT gate, eventually serving as the output into the Os signal. We can also see in Fig. 5 that this ring oscillator is integrated into an existing array of logic functions, containing 256 instances of our ring oscillator code (indexed from 0-255). This array of logic gates forms the fundamentals of our FPGA which will be the foundation of our PUF on the actual hardware which produces our desired challenge response pairs.

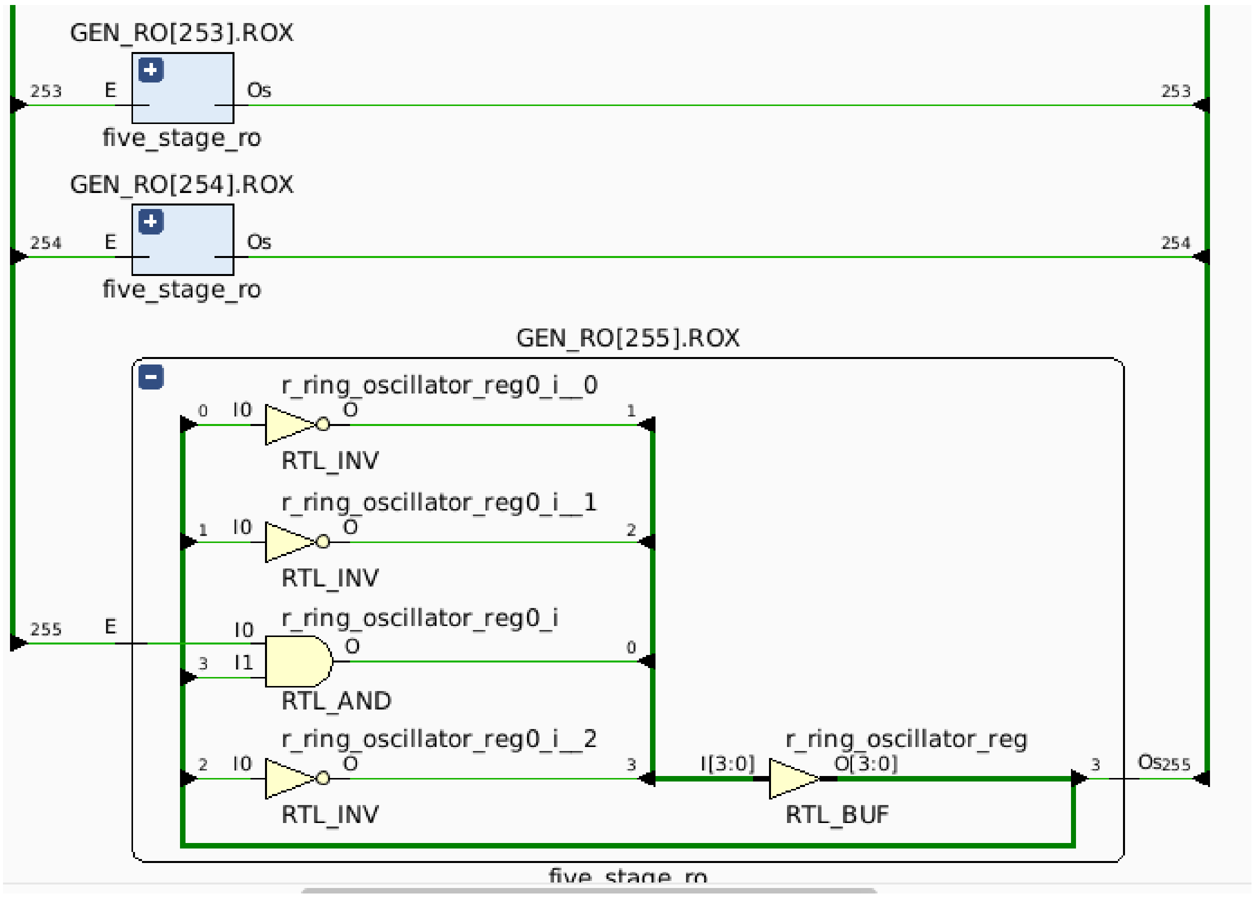
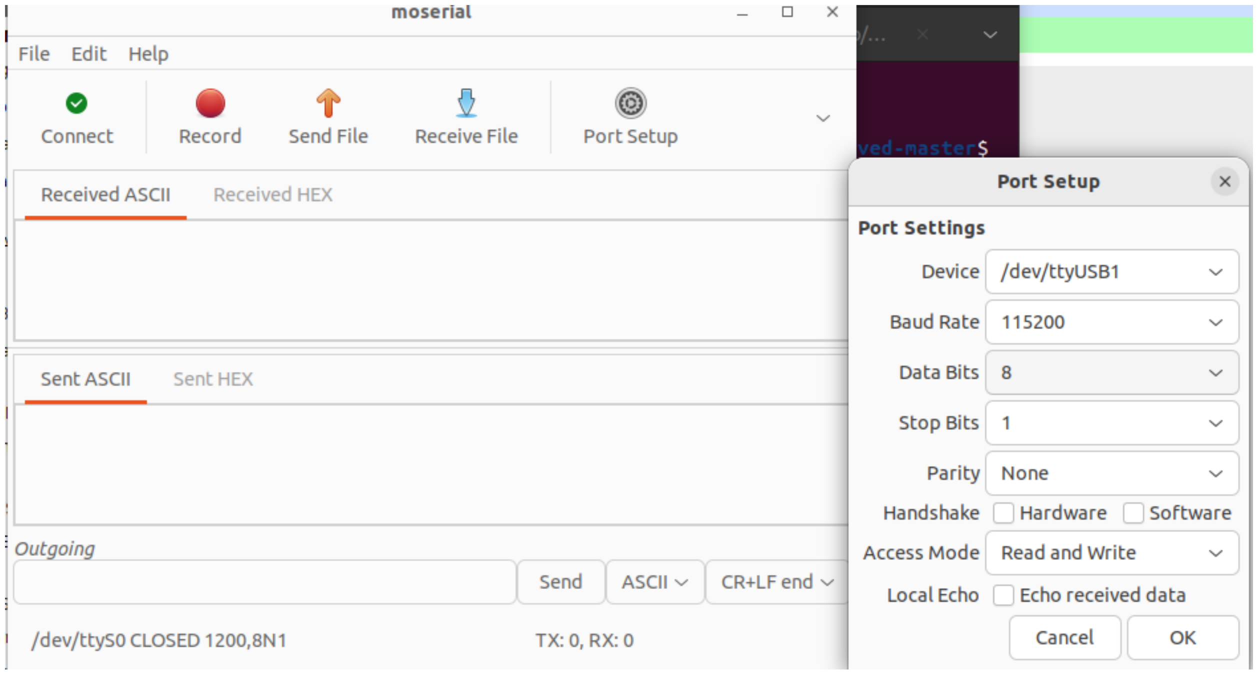


Figure 5. Vivado schematic of our ring oscillator VHDL code

Now that we have created the logic that is integrated into the pre-existing FPGA and verified its configuration with our schematic, we then programmed our Nexys A7 circuit board with this VHDL code. This allowed us to program the FPGA chip on our device and record its functionality which will serve to create the desired PUF output. After using the command “make all” in the PUF_HDL directory to compile the code and generate our bitstream files, we use the Vivado command “Program Device” and upload this bitstream to the board. Once this was completed, we proceeded to test our FPGA-based PUF on our device with the “moserial” analysis tool. Using the command “sudo apt install moserial”, we were able to install a moserial client to help us test the functionality of our PUF.



/dev/ttyS0 CLOSED 1200,8N1 TX: 0, RX: 0

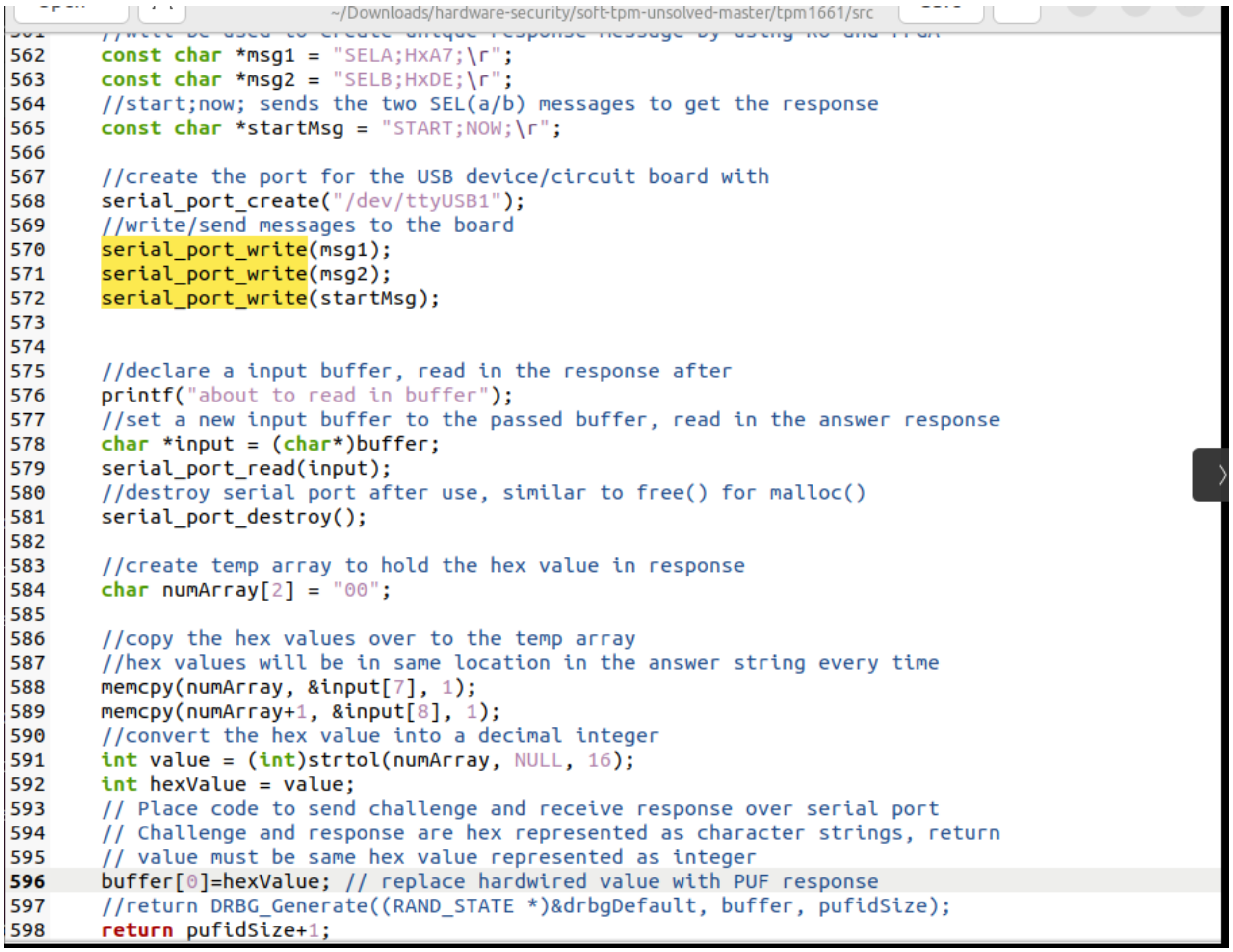
Port Settings

Device: /dev/ttyUSB1
Baud Rate: 115200
Data Bits: 8
Stop Bits: 1
Parity: None
Handshake: Hardware Software
Access Mode: Read and Write
Local Echo: Echo received data

Figure 6. Setup of moserial port settings

Fig. 6 displays the configuration settings we applied to our moserial client that allowed us to send challenge messages and receive responses. As referenced in Fig. 6, we set up our moserial client to connect to our device through the /dev/ttyUSB1 port. After configuring these port settings, we were able to test our circuit. We passed three challenge messages, “SELA;Hx55;”, “SELB;Hx88;” and “START;NOW;”. These challenges sent two messages containing the hexadecimal values 55 and 88, along with the command to start which is what directs the PUF to send the response. These hexadecimal values serve as the input to the Physical Unclonable Function, which has been programmed to map two hexadecimal inputs to a single hexadecimal output. When we send these challenge messages, we receive the message “ANSR;Hx00;”, verifying that our ring oscillator and PUF is functioning correctly. Once we construct our C program we will receive a non-zero response value, but the 00 is the correct response before creating this serial communication functionality.

Now that we verified the implementation of our ring oscillator and FPGA-based PUF by programming our circuit board, we move to completing the C program that allows us to send and receive CRPs from the PUF using the SoftTPM client-server relationship. For this program, we were assigned the responsibility of completing a single getPufid() function that reads in the challenges sent to the PUF, processes the response value, converts this value to a decimal integer and returns it. The server and client then facilitate this communication and the client will output the same hexadecimal value as was received by the program itself. This C code uses a serial communication protocol due to its space allocation benefits and simplicity in implementation. Serial communication requires much less physical space and wires as each signal bit is sent one at a time. Although this can sacrifice the speed of communication, we can make up for it by saving space on our devices in which space allocation is a scarce and valuable resource.


~/Downloads/hardware-security/soft-tpm-unsolved-master/tpm1661/src

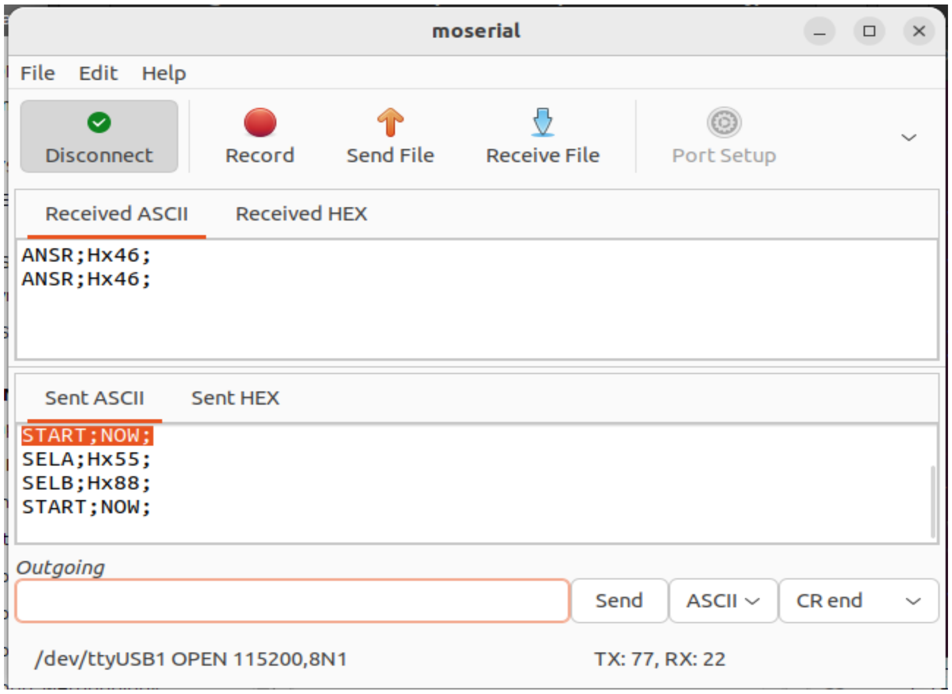
```
562     const char *msg1 = "SELA;HxA7;\r";
563     const char *msg2 = "SELB;HxDE;\r";
564     //start;now; sends the two SEL(a/b) messages to get the response
565     const char *startMsg = "START;NOW;\r";
566
567     //create the port for the USB device/circuit board with
568     serial_port_create("/dev/ttyUSB1");
569     //write/send messages to the board
570     serial_port_write(msg1);
571     serial_port_write(msg2);
572     serial_port_write(startMsg);
573
574
575     //declare a input buffer, read in the response after
576     printf("about to read in buffer");
577     //set a new input buffer to the passed buffer, read in the answer response
578     char *input = (char*)buffer;
579     serial_port_read(input);
580     //destroy serial port after use, similar to free() for malloc()
581     serial_port_destroy();
582
583     //create temp array to hold the hex value in response
584     char numArray[2] = "00";
585
586     //copy the hex values over to the temp array
587     //hex values will be in same location in the answer string every time
588     memcpy(numArray, &input[7], 1);
589     memcpy(numArray+1, &input[8], 1);
590     //convert the hex value into a decimal integer
591     int value = (int)strtol(numArray, NULL, 16);
592     int hexValue = value;
593     // Place code to send challenge and receive response over serial port
594     // Challenge and response are hex represented as character strings, return
595     // value must be same hex value represented as integer
596     buffer[0]=hexValue; // replace hardwired value with PUF response
597     //return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, pufidSize);
598     return pufidSize+1;
```

Figure 7. Full C code displaying the completed getPufid() function

Fig. 7 displays the implemented C code that builds our serial communication functionality. We begin with declaring the challenge messages in lines 562-565. The hexadecimal values of A7 and DE serve as the challenge message and the input to our PUF. We send these messages with the provided functions ‘serial_port_create()’, setting up the serial port to our device, and ‘serial_port_write()’ which actively sends the messages over the port. Importantly, we include the ‘/r’ carriage return character which tells the system to actually send this message. After creating the port and sending our challenges, we declare an input buffer which we set equal to the buffer that was passed to the function as a parameter, and use the ‘serial_port_read()’ function to read in the response from the PUF. We then use the ‘serial_port_destroy()’ function to close the serial connection after we receive the response. This process is similar to using free() with a malloc(), as we do not want to leave hanging ports open that are not needed.

After sending and reading our data, we must grab the hexadecimal value from our PUF response and convert it to a decimal integer. We create a temporary character buffer named numArray with space allocated for the two hexadecimal characters. We then used the C function ‘memcpy()’ to copy the character at a specific index into our numArray. The responses from the PUF will always be in the same format “ANSR;Hx??;”, with the ‘??’ being our hexadecimal values at indexes seven and eight. Therefore, we can use those indexes within our memcpy function to transfer those exact values to our numArray. Now that we have stored the desired values, we need to convert them from characters to actual hexadecimal values, then convert them to integers. We use the ‘strtol()’ function, passing the numArray and the value ‘16’ which represents the base we want to convert the strings to. We use 16 to represent the base of the hexadecimal system, then use the (int) casting function to convert these hexadecimal values to an integer. This integer value is then assigned as the first value in our buffer and returned. We can then run our ‘make’ commands within the TPM server directory to compile and use the TPM client to test the C code.

After completing the getPufid() function within our C program, we now utilize the client-server design of our SoftTPM client to execute the code and verify the accuracy of our PUF design along with our serial communication protocol. We begin in the tpm1660/src directory within our provided file system, and run the command ‘make’. This compiles our C programs and all other source and header files needed for our serial communication, and also creates a binary executable file named ‘tpm_server’ we can execute to start our server. We then navigate to the /tss1160/utils directory for our client functionality. We run the command ‘make -f makefiletpmc’ to generate the necessary binaries needed for communication protocol. We then run the commands ‘./powerup’ and ‘./startup’ to initialize the TPM client, and then use the command ‘./getpufid’ to execute our C function that sends and receives our CRP. This then sends our messages over the serial communication to our device, through our programmed FPGA-based PUF, and receives the output which should be the same as the response we received from our moserial testing client. We first tested our C program on the moserial client with the challenge hexadecimal values of 55 and 88, getting a response of 46. We repeated this command to ensure the stability and reliability of our serial communication, meaning the CRP could be repeated if it was given the same challenge.



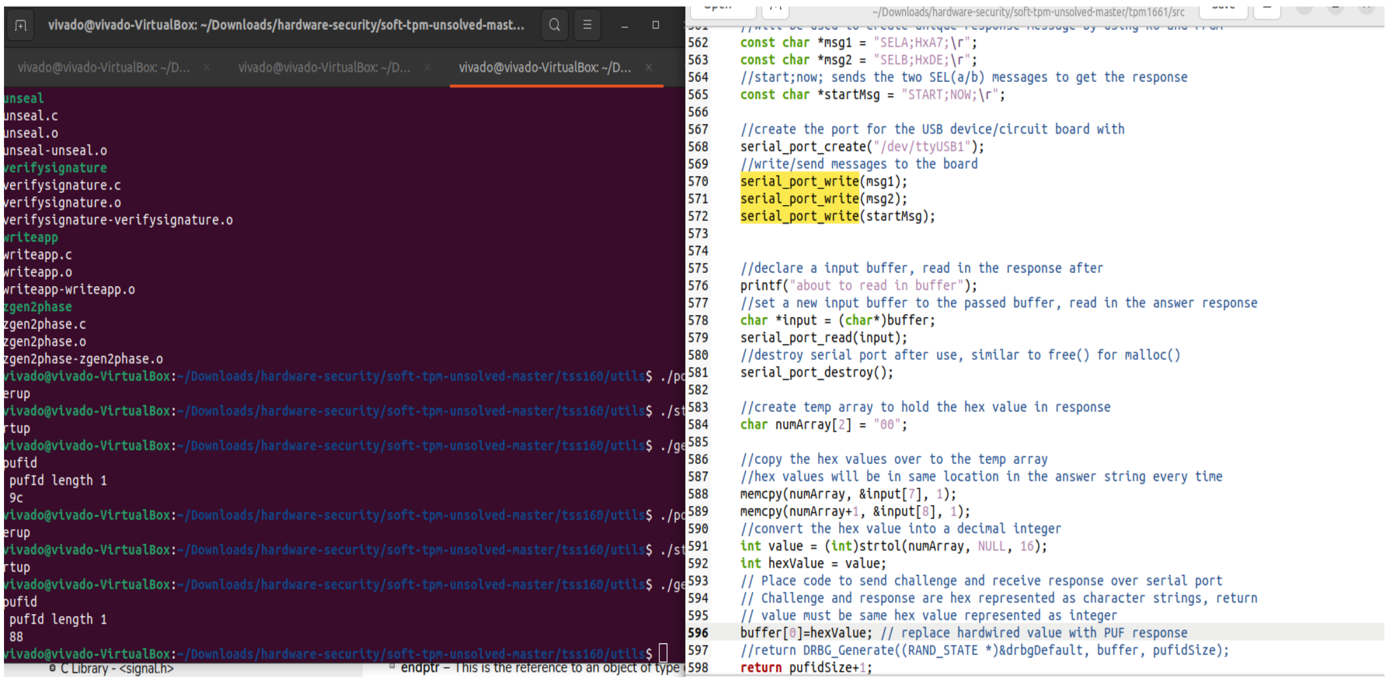
```
ANSR;Hx46;
ANSR;Hx46;
```

```
START;NOW;
SELA;Hx55;
SELB;Hx88;
START;NOW;
```

/dev/ttyUSB1 OPEN 115200,8N1 TX: 77, RX: 22

Figure 8. Proof of reliability and stability of our serial communication through moserial

Fig. 8 proves this reliability, as we receive the same answer of 46 after sending the same challenge messages twice over the PUF. We then use a new pair of challenge values to send over our TPM server and client, which we set as the values of A7 and DE as displayed in Fig. 7. We then executed the code through the client and server as previously mentioned and received our response within the client terminal.



```
vivado@vivado-VirtualBox: ~/Downloads/hardware-security/soft-tpm-unsolved-mast...
unseal
unseal.c
unseal.o
unseal-unseal.o
verifysignature
verifysignature.c
verifysignature.o
verifysignature-verifysignature.o
writeapp
writeapp.c
writeapp.o
writeapp-writeapp.o
zgen2phase
zgen2phase.c
zgen2phase.o
zgen2phase-zgen2phase.o
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./pc
erup
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./st
rtup
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./ge
pufid
 pufId length 1
 9c
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./pc
erup
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./st
rtup
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$ ./ge
pufid
 pufId length 1
 88
vivado@vivado-VirtualBox:~/Downloads/hardware-security/soft-tpm-unsolved-master/tss160/utils$
```

```c
const char *msg1 = "SELA;HxA7;\r";
const char *msg2 = "SELB;HxDE;\r";
//start;now; sends the two SEL(a/b) messages to get the response
const char *startMsg = "START;NOW;\r";

//create the port for the USB device/circuit board with
serial_port_create("/dev/ttyUSB1");
//write/send messages to the board
serial_port_write(msg1);
serial_port_write(msg2);
serial_port_write(startMsg);


//declare a input buffer, read in the response after
printf("about to read in buffer");
//set a new input buffer to the passed buffer, read in the answer response
char *input = (char*)buffer;
serial_port_read(input);
//destroy serial port after use, similar to free() for malloc()
serial_port_destroy();

//create temp array to hold the hex value in response
char numArray[2] = "00";

//copy the hex values over to the temp array
//hex values will be in same location in the answer string every time
memcpy(numArray, &input[7], 1);
memcpy(numArray+1, &input[8], 1);
//convert the hex value into a decimal integer
int value = (int)strtol(numArray, NULL, 16);
int hexValue = value;
// Place code to send challenge and receive response over serial port
// Challenge and response are hex represented as character strings, return
// value must be same hex value represented as integer
buffer[0]=hexValue; // replace hardwired value with PUF response
//return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, pufidSize);
return pufidSize+1;
```

Figure 9. Proof of serial communication with correct output in the client terminal (left)

Fig. 9 displays both our getPufid() function on the right and its accurate output on the left within our TPM client terminal. The returned response value of 88, which we also verified within the moserial software, is the response value to our challenge values of A7 and DE as shown in Fig. 9. Now that we have received the correct output from our C program, we accurately verified the design of both our ring oscillator and the serial communication protocol.

4. Results

Through the process of combining the design of a ring oscillator and completion of a C program to facilitate serial communication to our circuit board, we were able to successfully generate stable challenge response pairs that could be used as a form of authentication within a private key encryption scheme. For the design of our RO, we used an enable port, an output port, and a vector signal that held up to four values. We utilized three NOT gates to serve as inverters, allowing us to achieve stable oscillation which can be used to analyze the frequency of our FPGA-based PUF and verify the accuracy of its CRPs. Secondly, we were able to construct a C function that sent our challenge messages over a serial port to our device, and received a response from this same port to complete our serial-based communication protocol. This allowed us to generate stable, unique, and repeatable CRPs that can be used as a form of authentication and cryptographic key generation to harden the security of our hardware. We then used a SoftTPM client and server to facilitate this communication and produce output, using the SoftTPM function ‘getPufid()’ to retrieve the response’s hexadecimal value from the server. This allowed us to further verify the success of our ring oscillator design and correctness of our C program.

We can further analyze the results of our solution by using two common PUF quality metrics, namely reliability and uniqueness. Reliability, with respect to PUFs, refers to the concept of getting consistent responses from repeated challenges. Our ring oscillator and FPGA-based PUF was able to generate consistent output when we gave it the same input, meaning we had a reliable method of authentication and key-generation if we wanted to utilize the PUF in that manner. We also were able to get consistent responses for multiple different challenges, as we wanted to further verify the reliability of our design. Secondly, we can analyze our PUF with respect to its uniqueness to other similar PUFs on different chips. We compared our CRPs with other team’s and received mixed results, as some CRPs were the same and others had different responses for the same challenge. This meant a sub-par performance in uniqueness, as the goal of a PUF is to create completely unique responses to similar challenges, but given the success within the reliability metric, we concluded our design of both the RO and serial communication interfacing was successful.

Lastly, although we were able to achieve a successful design of the ring oscillator and were able to form a communication protocol, we were unable to compute the frequency of our RO or the gate delay of an individual inverter. We attempted the process of creating a debug core file and adding in the necessary constraints, along with running the proper synthesis operations to create the Integrated Logic Analyzer (ILA), but we were unable to properly generate this program. This prevented us from verifying the accuracy of our PUF challenge-response pairs as we were not able to measure the frequency which the PUF uses to determine the unique response to each challenge.

5. Discussion

We were able to successfully design a functioning ring oscillator and integrate it into an existing FPGA-based PUF, while also developing a C program to facilitate the serial communication to our device to generate our CRPs. However, we were unable to compute the frequency or gate delay within our RO, meaning we were unable to verify the results of our RO and PUF output, other than using the moserial tool to compare with. This lack of true verification by the hardware itself presents our solution with a few limitations on its applicability.

5.1. Bounds and Limitations

Due to our inability to calculate the frequency of our ring oscillator or the gate delay of a single inverter within the oscillator, we are unable to fully verify the accuracy of our CRPs and the output of our serial communication. This forces us to rely on the assumption that our hardware has not already been compromised and is functioning properly, which limits our ability to find any pre-existing flaws or threats within the hardware. This limits our solution's applicability to real world situations, as not being able to calculate the frequency of our oscillators leaves us vulnerable to an adversary's ability to alter the inner workings of our hardware. These alterations would often cause fluctuations to the frequency and gate delay, but not knowing these values at a standard level means we could not know when an attack was taking place.

Secondly, our solution is limited by the effectiveness and safety of our serial port communication and its ability to produce reliable results. We could not verify the accuracy or safety of this communication, and an adversary could have the potential to compromise the data being sent across this connection without being detected. This connects back to our inability to discover the frequency metrics of our design, as adversaries could perform several actions that compromise the security of our hardware without being discovered. This again limits the solution's applicability to larger, more complex situations and limits its versatility in the problems it can be used to solve without understanding the standard metrics of the solution itself.

5.2. Impact

Our solution represents the simple applications that ring oscillators and FPGA-based PUFs can be deployed for, and the impact on hardware security it can have. Our ability to communicate over a connection to our device and generate unique, yet repeatable responses to our challenge messages means we have created a design that can be used for authentication and private encryption key-generation to be able to validate the security of hardware. Hardware attacks are so dangerous due to their discreteness and difficulty in detection. Our solution provides an easy answer to this difficulty, as the CRPs of each PUF iteration will be able to determine whether or not the actual hardware has been altered since the original CRPs were generated. This can serve as a form of integrity verification, similar to message authentication codes within cryptography, that we can use to verify the security of our hardware from any potential attacks.

On the other hand, the poor performance of our solution with regards to its uniqueness points out the potential vulnerabilities to this form of security verification. Our solution proved to be reliable in producing stable and consistent results, yet also generated the same CRPs as other groups operating on different ICs, pointing out the potential vulnerability in using this technology. Even though an adversary does not have access to our specific IC, our experiments proved that they could possibly still obtain the exact CRPs of our system with another IC. As we can see in Fig. 10, PUFs can be used to authenticate a particular device using their unique CRPs and comparing them to an existing database. If an adversary were to compromise the CRPs of our device and attempt them on another device, our experiment showed it was possible to gather the same CRPs as another IC, meaning they could authenticate an untrusted device with a stolen CRP and remain undetected.

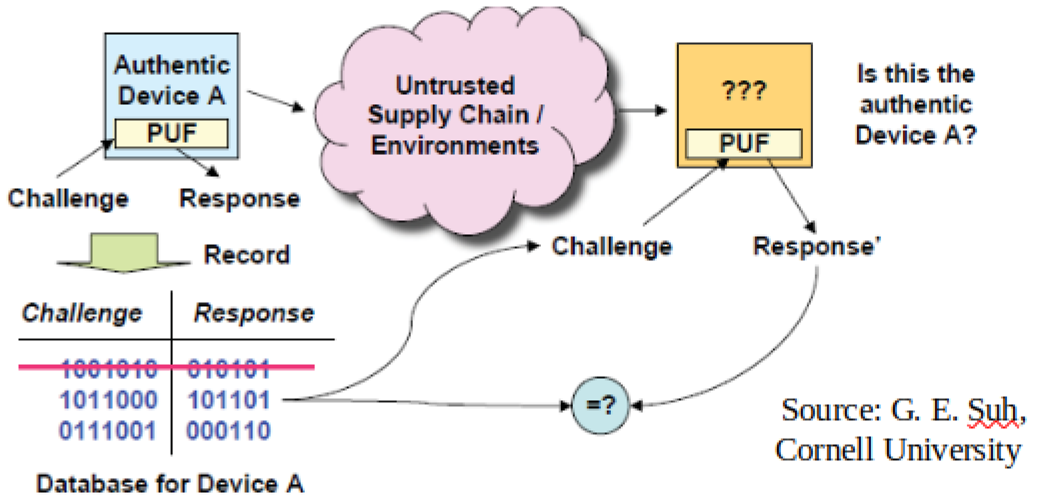


Figure 10. Depiction of use case for PUFs [7]

5.3. Future Work

We believe the future of this work should focus on completing the calculations of our frequency and gate delay, along with creating a more complex ring oscillator with five or more inverters rather than three. Firstly, being able to discover the frequency and gate delay of our circuit itself by using the Integrated Logic Analyzer (ILA) will allow us to measure and validate the security of our hardware, making sure there are no repeated anomalies taking place. Secondly, we would like to see a more complex ring oscillator be designed with a more complex chain of inverters. This would help to enhance the complexity of our design during implementation which could increase the safety of our hardware, but not need to increase the difficulty of its implementation. This could correct our poor performance in our PUFs uniqueness, as a more complex ring oscillator could help to make the PUF more unique to the IC it has been integrated on.

6. Works Cited

[1] “Hardware security breaches are virtually guaranteed | 2021-05-17 | Security Magazine,” *www.securitymagazine.com*. https://www.securitymagazine.com/articles/94815-hardware-security-breaches-are-virtually-guaranteed (accessed Aug. 06, 2023).

[2] eet.com, *CLB of an FPGA*. Accessed: 2023. [Online]. Available: https://www.google.com/url?sa=i&url=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F26046690%2Fneed-help-to-figure-out-how-the-clb-of-a-fpga-is-built-on-this-drawing&psig=AOvVaw0bRauQ0KwPYl1UaefoNqPy&ust=1691380188967000&source=images&cd=vfe&opi=89978449&ved=0CBAQjRxqFwoTCMi5sK2Qx4ADFQAAAAAdAAAAABAE

[3] “Minimizing Risk with FPGAs and Hardware-Based Security.” Accessed: Aug. 06, 2023. [Online]. Available: https://www.cisa.gov/sites/default/files/ICSJWG-Archive/QNL_SEP_21/Owl-article-minimzing-risk-hardware-security_S508C.pdf

[4] “An Introduction to Physically Unclonable Functions - Technical Articles,” *www.allaboutcircuits.com*. https://www.allaboutcircuits.com/technical-articles/an-introduction-to-physically-unclonable-functions/

[5] LogicBlocks, *Visualization of a ring oscillator*. Accessed: 2023. [Online]. Available: https://www.google.com/imgres?imgurl=https%3A%2F%2Fcdn.sparkfun.com%2Fassets%2Flearn_tutorials%2F2%2F1%2F6%2F29-oscillator-circuit_enable.png&tbnid=GQcu-x0f50TMpM&vet=12ahUKEwjH4dOXnseAAxUTUTUKHfyBC18QMygJegUIARDnAQ..i&imgrefurl=https%3A%2F%2Flearn.sparkfun.com%2Ftutorials%2Flogicblocks-experiment-guide%2F5-ring-oscillator&docid=gPbRcTWxOi5fGM&w=932&h=155&q=ring%20oscillator%20uses&client=ubuntu-sn&ved=2ahUKEwjH4dOXnseAAxUTUTUKHfyBC18QMygJegUIARDnAQ

[6] "Serial Communication - learn.sparkfun.com," *learn.sparkfun.com*. https://learn.sparkfun.com/tutorials/serial-communication/all

[7] G. E. Buh, Cornell University, *PUFs as a form of authentication*.