Injecting Data and Parity in Serial Communication with Field Programable Gate Array

Joseph Berman
Northeastern University
Boston, MA, USA
berman.jos@northeastern.edu

Abstract

Integrated circuits internally may transport data over a serial connection. Serial relies on a clock signal and a data signal. It is possible to inject data by microprobing into these integrated circuits and sending faulty data. However, this is easily prevented by having a checksum at the end of each serial transmission to ensure the data has been received correctly without errors. This can limit the functionality of an attack if it has such preventions.

A solution to this would be to inject our own parity check. To do so requires reading the serial data, calculating new parity data, and outputting the faulty data with the tampered parity.

To show this at a larger scale, this paper demonstrates this concept using microcontrollers connected over Serial Peripheral Interface (SPI) probed by a Field Programmable Gate Array (FPGA). In this scenario, the FPGA will read in bytes sent over SPI from one microcontroller, calculate a new checksum using the transmitted data and the data to be injected, then transmit the data and checksum faster than the next clock cycle of the controller.

Although this was unsuccessful in porting to the real world, this was successfully simulated using AMD/Xilinx Vivado. The limitations were issues with interfacing with the FPGA's General Purpose Input Output rather than the probing itself.

With a successful simulation, the goal of creating a timing attack with a corrected parity was achieved.

1 Serial Data

To simulate microproping an integrated circuit, two Arduino Uno were connected using SPI protocol. The reasoning behind this was to have a simple serial connection. Bits will be sent through the data channel one at a time on the rising edge of the clock channel. This was to best simulate the transfer of data through a serial connection in an integrated circuit.

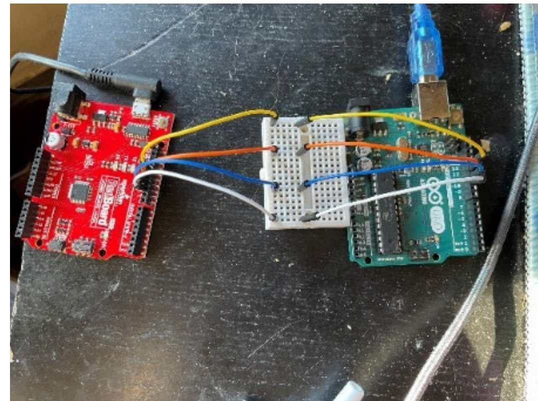

Figure 1: Arduinos connected over SPI.

SPI uses 4 channels: Clock, Controller Out/ Peripheral In, Controller In/Peripheral Out, Peripheral Select. In this scenario, only Clock and Controller Out/ Peripheral In will be probed as information is one directional. Peripheral Select is used to enable the current active Peripheral, but as it is the only Peripheral it will always be set to active in this situation.

A 5-byte transmission was chosen to be best to create a proof of concept. 4 Bytes would be for data transfer while one byte is for the checksum.

DATA (4 Bytes)				Checksum (1 Byte)
DATA[0]	DATA[1]	DATA[2]	DATA[3]	CRC

Figure 2: Datagram of transmitted data.

The checksum byte would be calculated by taking the XOR of the 4 data bytes.

$$\textbf{Checksum} = \textbf{Data[0]} \oplus \textbf{Data[1]} \oplus \textbf{Data[2]} \oplus \textbf{Data[3]}$$

One Arduino Uno was denoted as the transmitter. Using Arduino's programming language a basic transmitter routine was created. It would take a 4-byte vector, create the checksum, then transmit the 5 bytes in serial. The Arduino Uno's Maximum SPI clock cycle is at 8Mhz. The receiving Arduino Uno would accept 5 bytes of data and confirm that the data it received was correct by calculating a checksum.

```
{A1,B2,C3,D4,4}
{A1,B2,C3,D4,4}
{A1,B2,C3,D4,4}
{A1,B2,C3,D4,4}
{A1,B2,C3,50,A1}
Incorrect checksum
```

Figure 3: Arduino serial monitor outputting received data and throwing error when data is incorrect.

Having the checksum and limited clock speed, presents a challenge to inject data that would be accepted by the receiving transmitter. This would be the basis of what the FPGA is attacking.

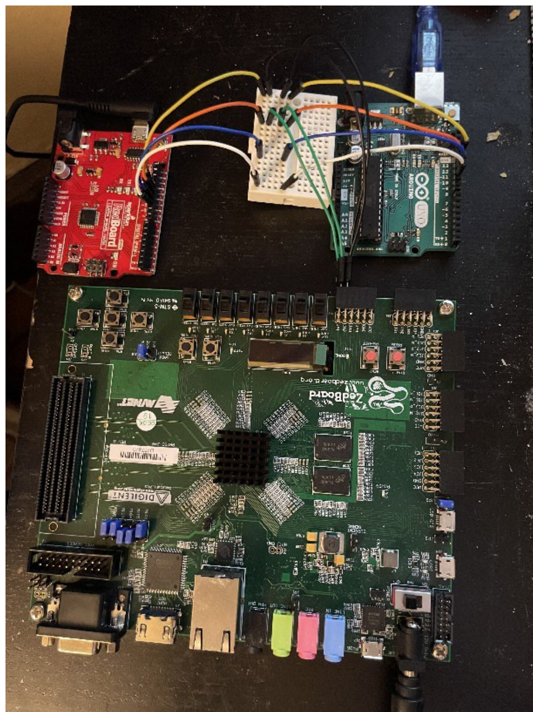

Figure 4: FPGA probing Arduinos. Green is connected to data, black is connected to clock.

2 Field Programable Gate Array

The chipset that was chosen was the Zynq-7000. This was provided by Avent's Zedboard, which is an evaluation board with multiple General Purpose Inputs Outputs. The maximum SPI speed of this board is 104 Mhz. These two qualifications make it an optimal board for cheap implementation and a basis for further research into a faster process.

The reason why an FPGA was chosen over a traditional processor is for its logic speeds. It can receive, process, and send data in bulk while a CPU must do so linearly. This will save clock cycles and be able to compute data faster than the Arduino can send another clock cycle.

The ZYNQ chipset has two sides, a traditional microprocessor and a programable logic side. The programable logic was used and was coded using Verilog. The Verilog module has 4 inputs and 2 outputs. The inputs are the FPGA's clock, the Arduino's clock, the Arduino's data, and a reset line. The outputs are the injected clock out and the injected data out. There are three total parts to the hardware implementation.

2.1 Input

The first part of the Verilog reads in from the probed SPI clock and SPI data channel (Controller Out, Peripheral In). The FPGA takes the first 24 bits sent and stores it into a shift register. At the same time, it keeps count of the number of bits that have been stored.

```
//INPUT
always @(posedge clk_in or posedge reset) begin
if (reset) begin
    shift_reg <= 24'b0;
    shift_pos <= 5'b0;
end else begin
    // shift data in
    if (shift_pos < 24) begin
        shift_reg <= {shift_reg[23:0], data_in};
        shift_pos <= shift_pos + 1;
    end
end
end
```

Figure 5: Input logic, receives input bits and stores it into shift register.

For its simplicity, despite its limitation for timing (which will be explained later), the 4th byte is the one that will be injected into the serial stream.

Once the shift register has been filled up it will send the data to the checksum module.

2.2 Checksum

The checksum module is its own hardware implementation. The nature of the project requires the checksum to be calculated as fast as possible. This is achieved by having them blocking. This means that the value will be calculated instantaneously. The module has two inputs and one output. The two inputs are the shift register holding the data from the Arduino and the other is the data that is to be injected into the SPI stream. The output is the return value of the checksum.

```
module CheckSum(
    input [23:0] shift_reg,
    input [7:0] inject_data,
    output reg [7:0]checksum
    );

    always @* begin
        checksum = 8'b0;

        checksum = checksum ^ shift_reg[7:0];
        checksum = checksum ^ shift_reg[15:8];
        checksum = checksum ^ shift_reg[23:16];

        checksum = checksum ^ inject_data;
    end

endmodule
```

Figure 6: Checksum logic, as seen the shift register and injected data are XOR and returned in the checksum value.

2.3 Output

The output part of the code consists of two loops to inject the faulty data. The output begins once three conditions have been met.

1. The input bit counter has reached 24.
2. The clock in is currently 0.
3. The data in is currently 0.

The initial condition is required as the output requires all the input data to be collected. The clock in and data in have to be 0 so the FPGA can inject into the probed channels without interference. Unlike the checksum data has to be managed on the FPGA's clock cycle to sync data sending. The first loop sends the injected 8 bits of data while the second loop sends the injected 8 bits of checksum.

```
//OUTPUT
always @(posedge clk) begin


    // Shift in data
    if (shift_pos == 24 && !clk_in && !data_in) begin
       //This is outputting 4th faulty byte and checksum
       for (i=0; i < 8; i = i+1) begin
           clk_out <= 1;
           data_out <= DATA_TO_SEND[i];
           @(clk);
           clk_out <= 0;
           @(clk);
       end

       //Sending out checksum
       for (i=0; i < 8; i = i+1) begin

           clk_out <= 1;
           data_out <= checksum[i];
           @(clk);
           clk_out <= 0;
           @(clk);
       end

        #1;
        shift_pos = shift_pos+1;
    end

    clk_out <=0;
    data_out <= 0;

 end
```

Figure 7: Input logic, receives input bits and stores it into shift register.

Once the three conditions have been met it will start the loops. The logic here is non blocking and will execute at every FPGA clk cycle. After the loops have been completed it will return to outputting 0 on the clk_out and data_out to not interfere further with the Arduino.

3 Simulation

To test the FPGA injection code, it was simulated using AMD/Xilinx Vivado. The chosen input data sequency was **[0xA1, 0xB2, 0xC3]**, and the faulty data that was selected was **[0xD4]**, which would create a checksum value of **[0x04]**. Testbench code was written to create this simulation. The Clock for the FPGA was chosen to be 10Mhz and the Arduino clock was chosen to be .1 Mhz.

```
initial begin
    clk = 0;
    forever #CLK_PERIOD clk = ~clk;
end

initial begin
    clk_in  = 0;
    forever #CLK_IN_PERIOD clk_in = ~clk_in;
end
```

Figure 8: Start of Arduino clock and FPGA clock.

Once the clock cycles have started an initial reset is sent to the simulated modules

```
// Stimulus
initial begin
    clk = 0;
    clk_in = 0;
    data_in = 0;
    //Reset
    reset = 1;
    #50;
    //Turn reset off
    reset = 0;
    #10;


    // data_in
    for (i = 0; i < 24; i = i + 1) begin
        @(posedge clk_in);
        data_in = DATA_TO_PASS[i];
        //#CLK_IN_PERIOD;

    end
    @(posedge clk_in);
    data_in = 0;

end
```

Figure 9: Simulate logic to send data in.

3.1 Simulation Results

The simulation showed a successful result. **Figure 10 and 11 in the appendix** show a successful run of a simulation. **Figure 10** shows the entire timing diagram which visualizes the Arduino clock in and data in. The data out, clock out, and FPGA clock are too small to see at this scale. **Figure 11** shows a zoomed in look at the timing diagram which highlights the successful output of injected data with a correct checksum.

4 Results

One additional part to the project that is currently unfinished is the implementation of mapping the GPIO to the Verilog module. Because of this there is no current real-world example of this working. Multiple assumptions are made to get a successful result.

The main assumption is that the peripheral can handle the fast clock speed that the FPGA is outputting. If it can't handle the speed of the output clock then data will be lost causing an incorrect checksum. An improvement to this is to change the byte that is being injected. By injecting on the 4th byte and immediately having to send the checksum byte, the FPGA will require 16 continuous cycles to send the full data. If injected on the 1st byte there can be issues with knowing when the later bytes will be received.

Assuming there is a constant clk speed from the Controller it would be more optimal to inject data on the 2nd or 3rd byte. This would allow the FPGA to send 1 byte of data in 8 continuous cycles of the FPGA clock then the checksum in a separate 8 continuous cycles. The benefit of this is allows for a faster clock that the controller is outputting while still being able to successfully input the data.

Although this was only successfully simulated, the logic and the concept of successfully injecting data remain. It is a great example of the ability to inject data and calculate a correct parity byte to limit the detection of an attack.

5 Conclusion

In all, this project was a great learning experience about using programable logic as a stealthy hardware timing attack. SPI was chosen as the protocol here was it was simple and fast. This was in hopes of making it similar to attacking an IC by microproping and attaching it to a FPGA. If the IC is sending data internally over a serial and a clock it is possible to inject data.

There are ways however to protect against this attack. The first being is to limit the maximum speed that the data can be received. If the FPGA is unable to send complete data faster than the next full cycle of the transmitter's clock or if the receiver is not capable of reading fast enough, this attack will be unsuccessful. Another would be to create a multibyte checksum. If the checksum if for example 2 bytes long, then it would require the FPGA to process and send 2 bytes faster than the transmitter can transmit a single bit. The single bit would cause the checksum to be incorrect which would raise a flag that the data transmitted is incorrect.

This was a great theoretical project that I plan on continuing after this submission. I found that I enjoyed working on this concept and enjoyed having to learn Verilog outside of a classroom or prescribed assignment. Being able to define my own project and solve the problem I created was very satisfying and made the project more personal to me.

References

[1]
"ZedBoard Zynq-7000 Development Board Reference Manual - Digilent Reference," *digilent.com*. https://digilent.com/reference/programmable-logic/zedboard/reference-manual (accessed Apr. 26, 2024).

[2] Arduino, "UNO R3 | Arduino Documentation," *docs.arduino.cc*, 2024. https://docs.arduino.cc/hardware/uno-rev3/

Appendix

Code: https://github.com/JosephBerman/fpga-injection

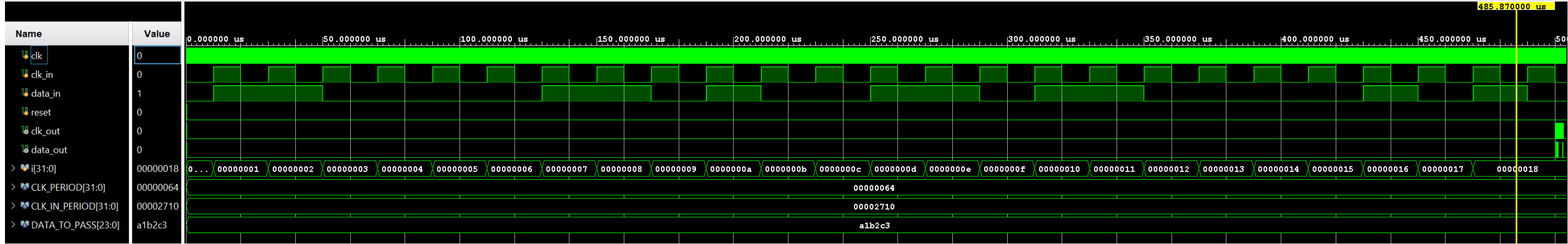

Figure 10: Full timing diagram

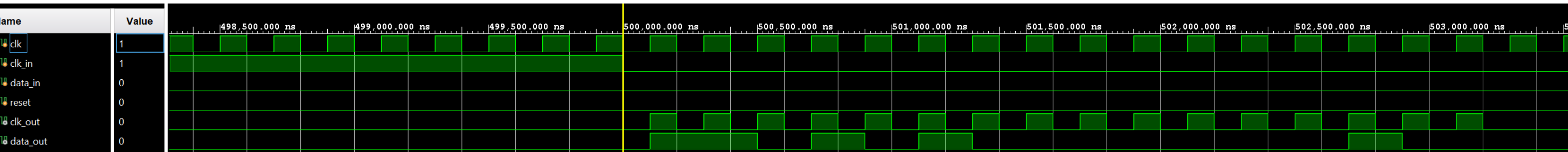

Figure 11: Zoomed in timing diagram to show injection. Binary output results in [0xD4, 0x04]