HDM-SPY

Kravitz, Samuel  
*Electrical Engineering and Computer Science  
Massachusetts Institute of Technology*Cambridge, MA  
skravitz@mit.edu

*Abstract*—I present the HDM-SPY, a proof-of-concept side-channel attack that exploits the electromagnetic radiation that are a byproduct of HDMI signal transmission. The system is implemented using an FPGA toggle on and off an HDMI stream to a 720px1280p monitor, and a Zynq RFSoC with an FM radio antenna to detect and decode the electromagnetic signature. Communication is achieved using On-Off keying modulation, capable of operating at bit rates of up to 1 kHz.

Keywords—RFSoC, HDMI, side-channel attack

# Introduction

This project is sort of a mashup between two projects. The first is the *Tempest-SDR* exploit, which uses the RF signal generated by the HDMI transmission to try and recreate whatever would be displayed on the screen of the connected device [1]. The second is the *RAMBO* attack, which uses the emissions generated by writing to RAM to leak information out of an air-gapped computer [2]. In this project, I combine these two approaches by replacing RAM with HDMI as the transmission medium responsible for generating spectral leakage. HDMI cables, which are widely used in many electronic devices, can still exhibit emissions that can be used to encode data, similar to the emissions seen with RAM. Next, I look for any RF signal from the HDMI transmission. Using similar methods as the Tempest-SDR project, my project detects, filters, and analyzes the RF emissions from HDMI transmission to decode the data being transmitted. Although this is just a proof-of-concept, the abundance of HDMI enabled devices makes this a very real vulnerability as already demonstrated by the *Tempest-SDR* project.

# Transmitting Side

## Physical Setup

This consisted of a RealDigital Urbana Board, which is based around an AMD Spartan-7 FPGA. From the board, I used an HDMI cable to connect it to the monitor I was using. In order to maximize the emissions from the HDMI signal, I stripped back the outer protective layer, and the internal shielding. Programming the board was accomplished via the USB/JTAG port.

## HDMI Signal Generation

In order to achieve a resolution of 720p by 1080p at 60 frames per second, a total amount of 750 x 1650 pixels need to be sent each frame to account for blanking and syncing periods, leading to a pixel clock of 74.25 megahertz. This is accomplished using Xilinx built-in clock wizard, using the 100MHz clock provided by the fabric to generate the required 74.25MHz, and a 5x 74.25MHz clock, which is used in the TDMS serializers discussed later.

The standard pixel clock is then fed into the Video Signal Generator Module, which uses it to generate the hcount and vcount values used for rasterization, as well as the necessary control signals like syncing and frame count. This block is parameterizable, and the user can define a variety of signals including all active, blanking, and sync widths, making it modular for different display configurations. The outputs from the block are then fed into the TMDS encoding blocks, along with the 8-bit pixel values.

A diagram of a process

Description automatically generated  
**Fig 1: Transmit Side Block Diagram**

TMDS (Transmission Minimized Differential Signaling) is used in HDMI to reduce bit transitions and maintain DC balance, improving signal integrity over long cables and reducing electromagnetic interference (EMI). TMDS encodes each 8-bit pixel value into a 10-bit symbol through two steps: transition minimization and DC balance correction. Transition minimization reduces 1-to-0 and 0-to-1 transitions to lower the likelihood of signal degradation, while DC balance correction ensures an equal number of 1s and 0s across transmissions. In my project, I chose the active pixel values to alternate between 0xFF and 0x00 with the aim of increasing the strength of the 74.25 MHz signal, but I am unsure as to whether this makes a difference, and could be an area to explore in future work.

Once the 10-bit TMDS pixel values have been generated, they go through the “kill” stage, which is where they are either passed through to the serializers or killed off to do modulation of the HDMI output. The logic for whether they are let through or not is discussed later. Next, they are passed to the TMDS serializers, which utilize Xilinx’s OSERDESE2 parallel-to-serial converter to prepare the signal to be sent through the HDMI port. The 74.25MHz drives the parallel input, and the 5x 74.25MHz clock drives the serial output. These outputs are connected directly to a differential signaling output buffer, which drives the differential outputs of the HDMI port from the FPGA to the monitor.

To actually transmit “useful” information, I use simple on-off keying to modulate the signal. For a proof-of-concept, I used a hard-coded value of “2024” in ASCII, along with some start frame and end frame bits. The bit period was done using a counter, and for each period, the TMDS encoding and pixel clocks were “anded” with the of the hard-coded value at that given bit index, thus allowing normal HDMI transmission when the bit was a 1 and squashing it when the bit was a 0.

# Receiving Side

## Physical Setup

On the receiving end, I use another RealDigital board, the RFSoC 4x2. I used an FM radio antenna connected to one of the ADC inputs on the board, and placed the antenna as close as I could to the exposed HDMI cable. Another area of future work could be to measure the effect of distance on the bit-error rate of the system. The RFSoC was connected via ethernet to one of the lab machines, allowing for me to interface with the device via a Python Jupyter notebook.

## Programmable Logic

The first step of decoding was to convert the analog signals picked up from the antenna into digital values. This is accomplished via the RF Data Converter IP block that Xilinx provides, which I configured to output IQ values. This block allows me to set a Numerically Controlled Oscillator (NCO) in software, which will mix down from the NCO frequency to baseband. In order to utilize the 491.52 MHz clock provided internally to the ADC, I set the ADC sampling rate to 1.47456 GSpS or 3x the 491.52MHz value. From there, I set the decimation to 10x, meaning that I will only pass on 1 out of every 10 samples. One important thing to note is that the NCO mixer is applied before the decimation. This required an AXI clock speed of 147.456MHz, which is generated by using the 11.52MHz clock output from the RF Data Convertor block, and using a clocking wizard to step it up to the required value.

Since the mixer brings the signal at the NCO value into the baseband, the next step is to apply a low pass filter. I used the TFilter website [3] to generate the coefficients for an FIR filter with a passband from 0 to 0.5 megahertz, and a stopband from 1 to 73.728 megahertz. The required number of taps was 373, and I implemented 2 filters using the Xilinx FIR compiler, one for the I values and one for the Q values.

Next the values were fed into a piece of IP we developed as part of a previous lab, called the TLAST Generator. The purpose of this module is to batch 65,536 samples of the IQ data, concatenate them together, and provide a TLAST signal to make it AXI-Stream compatible for the downstream components. From there, they were passed to another self-implemented module, which would take in the values, split them into the separate I and Q values, square them, and sum them, creating a running sum of all the values in the 65,536 batch. This was a crude implementation of the root mean square method, essentially calculating some function of the magnitude of that batch of data. This value was later used by the Python notebook to decide whether the HDMI signal was on or off.

A diagram of a diagram

Description automatically generated  
**Fig 2: Receiving Side Block Diagram**

From the split-square-sum (SSS) module, the output was fed into an AXI-Stream FIFO, which allowed us to do clock domain crossing from the faster 147.456MHz clock required to drive the modules upstream of the FIFO to the slower 100MHz clock which drove all the downstream logic required to interface with the processing system. On the other side of the FIFO was the Direct Memory Axis (DMA) block, which was used to pass data from the programmable logic fabric to the processing fabric quickly.

## Software

Implementing the software required to complete the system is relatively simple compared to the rest of the project. First, I created a new folder and uploaded the bitstream and the block diagram overlay that are generated from Vivado. Next, I created a new Jupyter Notebook to actually interface with the processing onboard the RFSoC.

The notebook consists of 5 total blocks. The first block imports the pynq library, and the xrtfdc library, which is used to interface with the RF data converter. It then resets the programmable logic, imports the overlay, and instantiates the DMA and RF data converter objects. The next block simply defines which ADC tile and block we are using, and sets the NCO to 0 as a test. The 3rd block then sets the NCO to the actual 74.25 MHz value required.

The 4th block is where the actual decoding of the on-off keying occurs. First, I declared the bit period and the threshold of the magnitude. The bit period should match whatever is implemented on the transmitting FPGA, while the magnitude needs to be determined by some trial and error to get accurate results. Next, I use the allocate function from the pynq library to create an area of memory within the DMA and assign it a handle. Then to read the data from the PL side, I simply start a DMA transfer, and using the wait() function, the transaction will stop once it receives a TLAST signal. I am then able to get the value by indexing into the zeroth index of the handle corresponding to the section on the DMA.

To actually decode, I implemented a quasi-state machine, which waits for the HDMI signal to go high, and then times how long it stays high/low and prints the corresponding bits received by rounding the time between transitions and then dividing by the bit period. Finally, once the end frame sequence is displayed I copy and paste the received sequence to the fifth block. Since on the transmitting side, I transmit the data little-endian in both bits and bytes, the 5th block simply reverses the bit sequence from the 4th block and converts to ASCII.

# Evaluation and future work

## Evalutation

In terms of evaluation, I don’t have that much data, as due to timing constraints I was unable to fully test my system, however, I was able to transmit the 32-bit sequence with bit periods of both 1 second and 0.1 seconds. It is my expectation that the maximum baud rate is actually much higher, as reading in the magnitude from the DMA takes on the order of less than 1 millisecond, which means that in theory, the system is capable of transmitting data at rates on the order of kbps. I also was not able to test with different antenna distances, but the magnitude difference between an “on” and an “of” signal at very close range was ~100x, implying that there is potential for detecting the bitstream from further distances.

## Future Work

As mentioned previously, there are multiple areas for potential future work. The first area would be evaluating the system in a similar way to the *Rambo* project, where they looked at how different bit rates affected the corresponding bit-error rates. Next I would test how antenna distance affected the ability to discern the signal, to determine a maximum range of the attack.

An additional thing to test would be to look at the magnitude of the peak at the harmonics of the 74.25 MHz clock rate, as those might also be useful for detecting whether the signal is high or low. For both of these, one aspect that may have played into the simple testing I did was the lab environment, which contains many monitors, most of them connected via HDMI to other devices. While those cables were likely shielded, I would be interested in testing in a less noisy environment.

A graph with a green line

Description automatically generated

**Fig 3. FFT Plot of Filtered Signal:** This shows the “off” state of the HDMI signal after going through the low pass filter. The “on” state plot is similar, however the magnitude of the signal at baseband it 100x the value

Another area of improvement is moving even more processing to the logic side of the RFSoC. I think that it is possible to implement the on-off detection that is currently implemented in software into the hardware, and have that module write to the DMA, so that on the software side, all that is required is to simply poll the DMA to receive the corresponding bit. This would likely help the system approach the kbps bit rate I mentioned previously.

Finally, as mentioned before I alternated the pixel data between 0xFF and 0x00 with the intent of trying to maximize the signal at the base 74.25MHz frequency, a potential area of exploration is how changing the pixel data changes the magnitude, as maybe it is possible to encode multiple bits in a single frame (i.e. 0x12 gives a magnitude of x, 0x34 gives a magnitude y, etc.). This would likely be implemented with quad-keying, where the amplitude would represent two bits at a time, allowing for double the data rate for a given bit period.

All of the code for this project, including the HDL for the HDMI signal generation, the Vivado project used to generate the overlay and bitstream for the RFSoC, all the self-implemented IP, and the Jupyter notebook are available at https://github.com/samuelkravitz/RFSOC.

##### Acknowledgment

. I’d like to thank Joe Steinmeyer for helping me understand some of the DSP side of things and being a rubber duck for some of my thought process, and Andi Qu, for his extremely helpful piazza post which enabled me to configure the RF Data Convertor and set up the correct data pipeline in order to read accurate values from the ADC.

##### References

1. Marinov, Martin. TempestSDR. GitHub. Available at: <https://github.com/martinmarinov/TempestSDR>. Accessed December 9, 2024.
2. Guri, Mordechai. *RAMBO: Leaking Secrets from Air-Gap Computers by Spelling Covert Radio Signals from Computer RAM*. arXiv
3. <http://t-filter.engineerjs.com/>