F(PGA) Tour : Evading reverse engineering

Intro

Field Programmable Gate Arrays (FPGAs) have many characteristics that enable them to serve as the root of trust for medium-security settings. They are very difficult to reverse engineer, are easily programmable, and are relatively cheap. These characteristics make them an ideal location to hide and protect sensitive information. In a very adversarial environment, it is helpful to think defensively, and in this regard FPGAs have a role to play. This presentation presents an implementation that discusses some work in defensive planning, why FPGAs are an obvious location to protect information, and showcases the strengths of an FPGA in obfuscation and protecting information.

In a digital system, there are standard and obvious attack pathways. Memory is an easy target in digital systems, because that is where programs are stored. Microprocessors and microcontrollers are obvious targets in digital systems, because they run code. By examining the memory used by microprocessors, the skilled reverse engineer can bypass security by modifying the memory contents, or by changing the code. FPGAs don't have this handicap. The chief reason that FPGAs are a good location to protect information is that they are ridiculously complex to reverse engineer (RE). The FPGA configuration bitfile is a description of how to configure the FPGA. These configuration bitfiles are complicated and proprietary data structures that are difficult to decode.

FPGAs also have other advantages: they are cheap alternatives to custom ASICs. They are relatively easy to design, with the main difficulty being learing the cryptic hardware description languages such as Verilog or VHDL. They are awesome at bit manipulation. For these reasons, they make an excellent source of glue logic and computation logic in a circuit, and in our case, providing an obscuration technique to hide and protect sensitive while making it difficult to reverse engineer.

Difficult to Reverse Engineer

Many people have discussed reverse engineering an FPGA bitfile. It has been a frequent topic on the internet. However, many people don't realize just how difficult it is. I will be discussing the process to show the difficulty level.

To begin with, there are intentional impediments to retrieving the bitfile. Many of the current FPGAs have the option of either encrypting the bitfile data (if the data is stored external to the FPGA), or will electronically prevent you from reading the configuration bitfile data out. The first step can be a hurdle.

Once you have correctly retrieved the FPGA configuration bitfile, the next step is to decode it. The FPGA configuration bitfile is a proprietary format of thousands of millions of binary bits. In contrast, microprocessors have a published instruction set. Each manufacturer publishes all opcodes and the binary format to represent each instruction. With RISC-type processors, each instruction is on a 16- or 32-bit boundary. With more complex instruction sets (Intel being the absolute worst), each instruction still has a byte boundary, but can be represented in anywhere from 1-15 bytes depending on the instruction. These instruction sets are all published and publicly available. Documentation and wiki articles on instruction sets abound. However, information on FPGA configuration bitfiles is intentionally protected and kept out of the public domain. They will probably present you with a cease and desist order if they learn of you trying to RE their bitfile.

The information in a bitfile is also not in a consistent format. As an example, with microprocessor instruction sets, the instructions are at least on a byte boundary. This is not true

of FPGA bitfiles, rather each element in an FPGA can take 1 to thousands of bits to configure it - all arbitrarily placed. A 4-input look-up-table (LUT) can take 16-bits of the bitfile. A mux could take 5 bits. An embedded core could take hundreds to thousands of bits to describe. There are numerous types of resources that a bitfile needs to describe. As an example, Figure 1 shows a Configuration Logic Block (CLB) for an early Xilinx XC4000 FPGA. How many bits do you think it takes to describe that interconnect? The developers didn't try to restrict the format to 8-bit boundaries, or make it convenient for someone to understand.

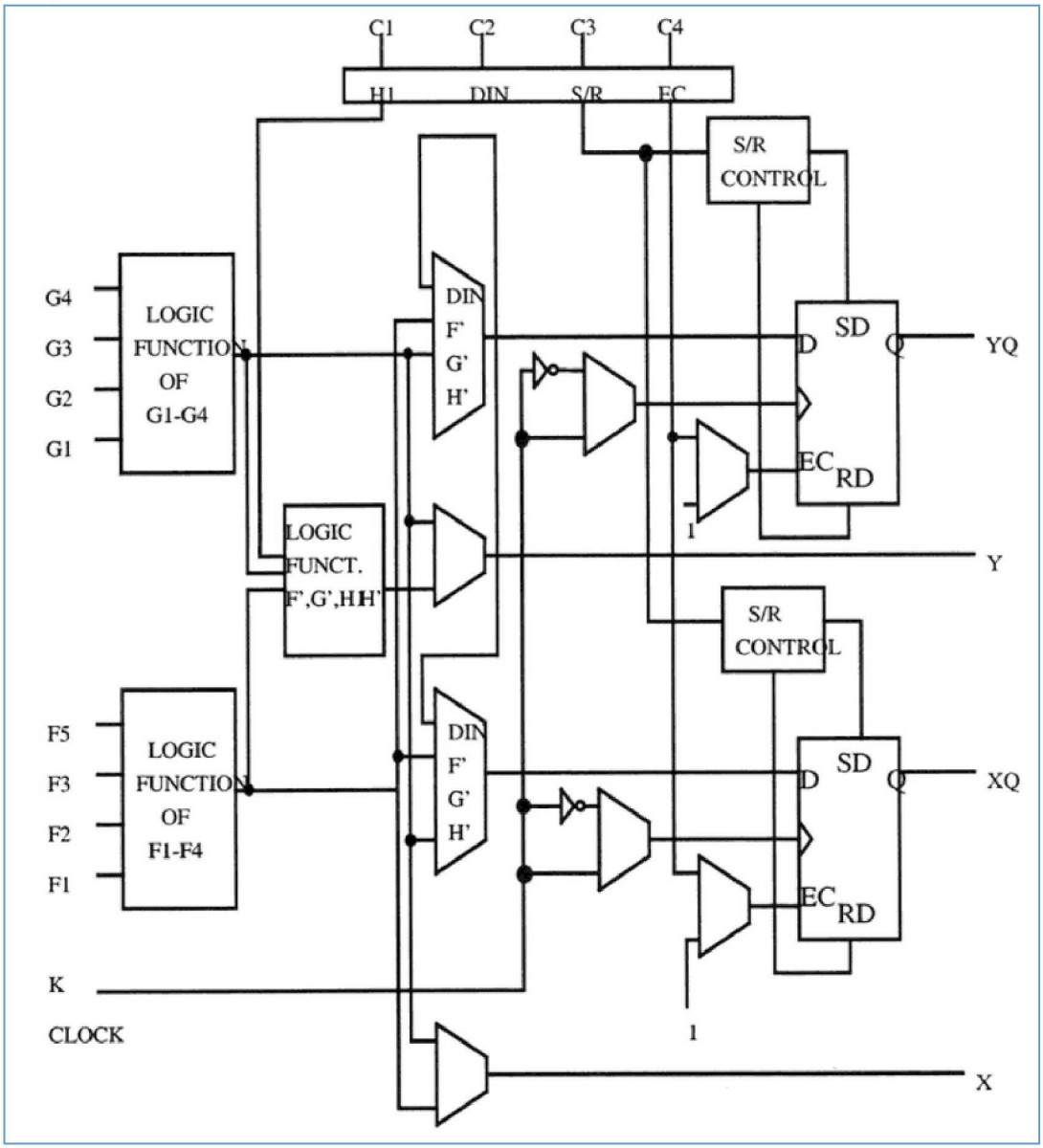

Figure 1 - Xilinx XC4000 Configuration Logic Block (CLB)

Another challenge in reverse engineering an FPGA is that it is not possible to only RE part of a bitfile, rather, one must RE the entire file. When reverse engineering software, the skilled engineer can quickly target the section of code that he wants to understand or modify. He can then spend time in a small section, and ignore the bulk of the remaining code. When reverse engineering an FPGA bitfile, the engineer must reverse engineer the entire bitfile. Each bit describes part of a circuit. Every bit has the potential to be something important or critical.

Also, even though an FPGA design never consumes the entire fabric of an FPGA, the economically minded engineer attempts to buy the smallest and cheapest FPGA that still meets his needs. Therefore, a majority of the FPGA bitfile will be filled with meaningful design information.

Yet another difficulty about reverse engineering an FPGA is that the process doesn't scale. If you have reverse engineered a bitfile for an FPGA, you have the bitfile for a single FPGA. Specifically, you only have the bitfile for THAT package for THAT part number for THAT family of FPGAs. Every variation has their own bitfile layout. It is not very cost effective to reverse engineer an FPGA configuration file.

How would you go about reverse engineering a configuration bitfile? The most straightforward way to do this is to use the brute force approach. You would make a design that uses a specific pin, and then create a simple design, such as an AND gate. You create the bitfile and observe what is generated. Then you repeat the process but modify the design so that it has a NAND gate. Then you do a DIF of the two generated files and make a guess as to what was changed and why. Then you iterate ad nauseum with incremental changes such as changing the pin that was used, or the I/O voltage, or the flipflop reset type, or a whole myriad of changes to slowly determine what each of the bits configure. The problem with this is that there will probably be a huge number of bits that you don't have the ability to set, or don't have the ability to affect through your input design files. There is a host of obscure settings in a bitfile that only the designers truly understand. The brute force method will get you partially there, but it definitely won't get you all the way.

The next step would be to reverse engineer the bitfile generation software. There are three major FPGA designers: Xilinx, Altera, and Microsemi. Microsemi is for the lower-end devices. (It is the one I used for this presentation.) Libero is Microsemi's configuration bitfile design tool. It sits like a fat Buddha on my hard drive at a whopping 6.94 Gig, contains 27,526 files, and 3,495 folders. You could theoretically reverse engineer this, but I wouldn't want to do it. It is massive and complex. The Xilinx and Altera compilers are an order of magnitude bigger in size and complexity. The advantage is that if you do this, you will probably have reverse engineered every bitfile that the tool can generate. Win. Unfortunately, it might take you until the parts are obsolete. Lose.

Most likely if you were going to do this, you would use a combination of the two. Step one would be trying the brute force to get reasonably down the path, and then boot up some IDA Pro and see how much farther that gets you.

Let's discuss what the forward process is in generating a configuration bitfile so that we can understand how we would do it in reverse. Generating a bitfile is a complicated process. In order to create a bitfile, you first start with your design, coded in a Hardware Description Language (either Verilog or VHDL). From there, you use a compiler to parse the code into something that makes sense for the computer to use. The next step is to synthesize the code into logic representation, which is known as register-transfer level (RTL). In digital circuit design, RTL is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals between hardware registers, and the logical operations performed on those signals. The next step is to take the RTL and map it to an FPGA fabric so it can be implemented on that FPGA. This step also includes making sure that all of the logic is appropriately routed through the FPGA. The last step is to take that mapping, and create the correct bitfile for that FPGA. Voila. A working bitfile!

Taking the process in reverse is an even more complicated process. None of these steps exist in the wild, so I am going to make-up names for what the steps would be. The first step is to decode each bit into what it represents – a look-up table (LUT), a connection point, a setting on a flip-flop, etc. From there, you need to "desynthesize" the mappings and settings and map it into a binary representation of what the bits do. The "decompiler" would take the binary representation and make it into a circuit. At this point, the circuit would still be very

difficult to understand. Even small circuits like an adder or a mux are complicated. Larger circuits like state machines or crypto algorithms would be unwieldy and time-consuming to decode. One huge difficulty is that even though you have somehow painstakingly and magically reverse engineered every bit in the bitfile, and you have developed a magnificent tool that can then turn that information into a logic circuit, you still have the difficult process of reverse engineering the digital design, which is still tedious and difficult, but orders of magnitude easier in comparison. A typical digital design has multiple stages of logic calculations separated by clocked flip-flops. Those logic calculations could be descriptions for state machines, could be arithmetic logic units (ALUs), could describe a Rijndael s-box, or could be a whole myriad of other things. It would be nice if there was some sort of "demystifier" that could recognize chunks of logic and give the reverse engineer some guidance as to what the globs of digital logic are – adders, muxes, address decoders, etc. Eventually the demystifier would produce an output that looks like something the original designer started with. At this point, you could finally start the process of reverse engineering the circuit to determine functionality.

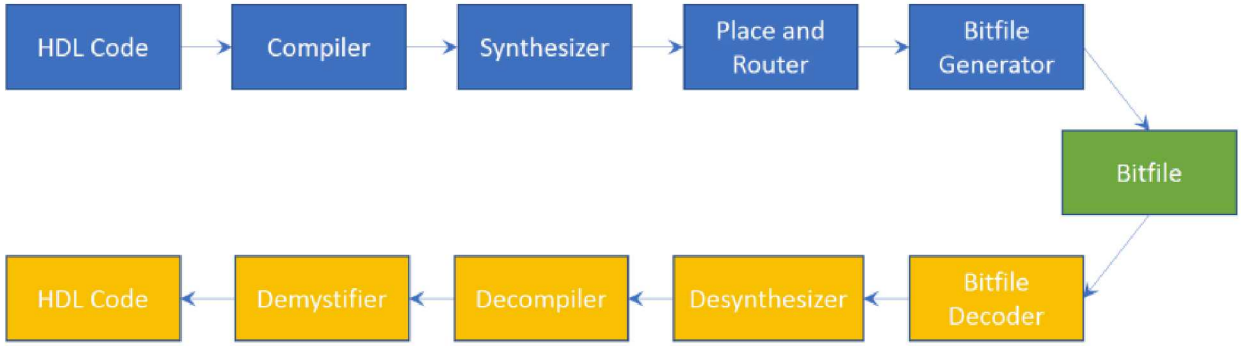

Figure 2 - Bitfile Processes

Around 2010, there was a tool called "Debit" that was created to reverse engineer Xilinx chips. The goal was for it to be the decoder and desynthesizer and create a text representation of the original FPGA circuit. The user would then have to take the text files and play decompiler to turn them into an analyzable circuit. I found the tool to generate incorrect interpretations, although I may have had an early version, or tried it on an FPGA that the program wasn't correctly programmed for. In my experiments, it returned large sections of incorrect data. It has been nearly 7 years, so there have likely been improvements.

For all of the reasons I have outlined, reverse engineering the design from an FPGA is difficult. Locating sensitive information in an FPGA is an effective mechanism to protect that information.

Our implementation

For this presentation, we have created an example implementation. The goal of the implementation is to show that you can do some interesting things with an FPGA to protect sensitive information. The design capitalizes on what FPGAs do well – bit manipulation. It is a very simple design, and so it is very fast, and very effective at what it does. This FPGA design was implemented as the secret keeper for the badges that my group brought to the conference this year.

This implementation uses obfuscation, which is different than encryption. Obfuscation is a process applied to information to intentionally make it difficult to understand without knowing the algorithm that was applied. Encryption is a process applied to information that, even knowing the algorithm applied, requires a secret or a key to understand it. Our example is

difficult to understand because the algorithm is unknown, not a key, therefore it is obfuscation, not true encryption. Our example is fast and requires only one clock cycle. Encryption would take much longer.

Another thing to note is that this demonstration establishes how easy bit manipulation is for an FPGA. On a software system, this code would probably dozens of lines of code and take hundreds of clock cycles. For an FPGA, it can be done in one cycle.

The implementation has two parts: the password authentication module, and the memory encoder/decoder module. We will discuss both in detail.

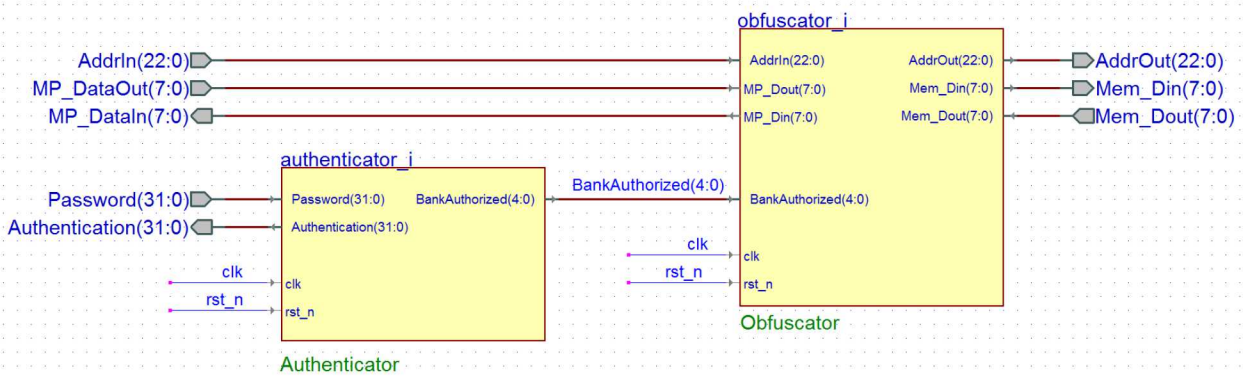

Figure 3- Implementation Block Diagram

The implementation is for an application similar to the dcdark.net quests. Once the user enters the correct password, then the decoding mechanism for the next section of code is enabled, and all other sections become encoded. To get access to other sections, the user must enter the correct password for the next chunk of code memory. Until the proper block is unlocked, the code memory is obscured in a way that would be very difficult to reconstruct. Once the decoding for the block of memory is enabled, this implementation doesn't care if you can reverse engineer the obfuscation mechanism for that particular block. It has already been opened, so is no longer protected. The details of the obfuscation for each block of memory is different. This is easy on an FPGA.

The hardware circuit for the demonstration board needs to be explained. In our implementation, a microprocessor is connected to a memory through an FPGA. The FPGA memory is partitioned into blocks that each have their own password, and their own obfuscation method.

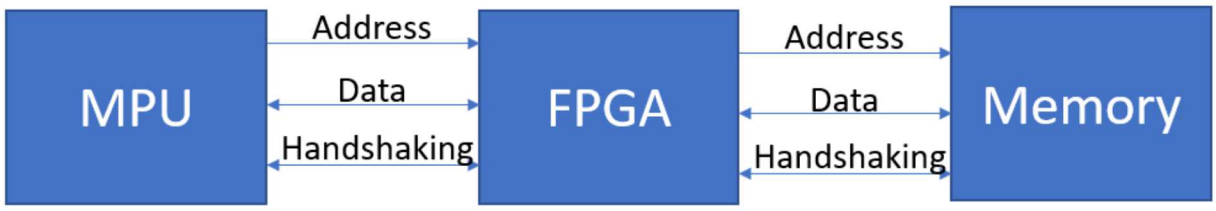

Figure 4 - Circuit Block Diagram

When a proper password is presented to the FPGA, all the addressing to that memory block will be correctly decoded. The data will also be correctly encoded or decoded.

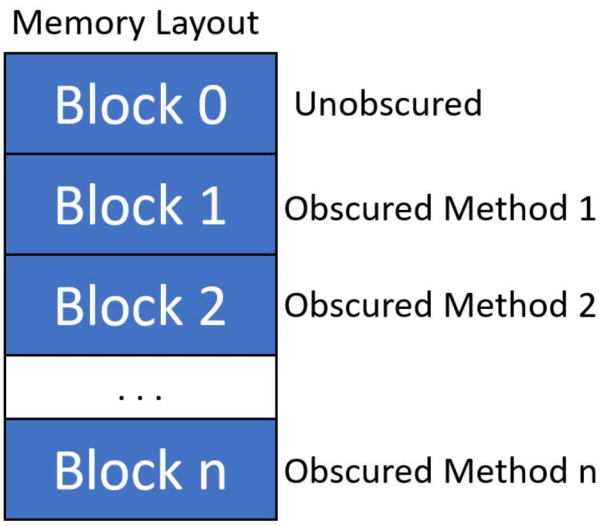

Figure 5 - Memory Layout

This implementation example was used on our own Defcon demo badge. Each block holds the code for a quest or puzzle. When the user believes he has the correct answer, he enters the password. A 32-bit hash is constructed from the answer, and is used as the password for that block. The password is sent to the FPGA. If the password is correct, the encoder/decoder is set to correctly interface to that particular block in memory. At this point the user is then provided access to the next puzzle.

Password Authentication Module

The first module is the password authentication module. All of the most sensitive information is held here. The function of the authenticator is to hold a list of passwords, one password for each block. For our implementation, the password is a 32-bit hash of the answer for the current quest. The authenticator returns a value when it is presented with a password. The return value is the jump address for the newly decrypted block of memory.

The Verilog code is presented blow:

```
always @(posedge clk) begin
        BankAuthorized = 6'h00;
        case (Password)
                32'h00000000 : BankAuthorized = 6'h00;
                32'hfafab0b0   : BankAuthorized = 6'h01;
                32'haabb1234 : BankAuthorized = 6'h02;
                32'hdeadbeef  : BankAuthorized = 6'h03;
                32'hcafebabe  : BankAuthorized = 6'h04;
                32'h2345abcd : BankAuthorized = 6'h05;
        endcase
        Authentication = {8'h00,BankAuthorized,18'h0};
end
```

Figure 6 - Authenticator Verilog code

For our implementation, a password is presented to the FPGA. It then returns an authentication value. The authentication value is used as a nonconditional jump address. The pseudocode would be:

```
SEND password_to_FPGA
READ authentication_value
JMP authentication_value
```

Figure 7 - Authentication pseudocode

This code removes any ability to change a compare/jump code section to a “NOP slide”. There is no comparison or choice in the software. The comparison and the branch are in the protected part of the FPGA where it can’t be discovered or reversed.

Joe Grand teaches a class in hardware hacking. He has a demo board used in the class to teach the principles of hardware hacking. The student’s goal is to reverse engineer the design and make modifications that enable a Simon game. There are three common solutions: one is to modify the memory to have the correct password, another is to read out the source code, find the comparison, and rewrite the code, skipping the comparison/branch, and a third solution is to use clock glitching to skip the comparison. In our implementation, you can’t do any of these solutions. You can’t modify the memory, you can’t rewrite the code and remove a comparison/branch instruction, and there is no branch instruction to glitch past.

For our implementation, if the password is not in the list of approved passwords that are stored in the FPGA, then the FPGA reverts to encoding and decoding from block 0 (which basically has no obfuscation). The return address for an incorrect guess is all 0’s. What that means, is when you get to the JMP #authorization_value, the code basically does a JMP to RESET_VECTOR at address 0x00000000, and resets the microcontroller.

The password in our implementation is a 32-bit password. For the example, that is an appropriate length. Bruce-forcing the password would require on the order of 4 billion guesses. That’s obviously a large search space, but not ridiculously unreasonable. Obviously not something you would use for banking or badge access codes to NORAD. If you want to limit the brute-force opportunities, there are a couple of deterrents. The implementation could include a 128-bit password, increasing the guess space to absurd levels. The implementation could also add a limited try count that invokes a penalty if too many passwords are attempted.

For our implementation, if the password is wrong, we return a 0, which is an indicator that the guess is wrong. However, sometimes it is useful to not give any indication of failure. In that case, the FPGA could simply return a random value, but not enable the decryption. The JMP would be sent to a random location in a decoded section, where the microprocessor would then start executing random assembly code. It would be interesting to see how long it takes for the microcontroller to hang itself.

At this point, I need to add a “justification” for the “encryption” used. The encoder/decoder does NOT use an approved encryption algorithm. This is simply a toy obfuscation method that is mathematically based, but not cryptographically sound. It gets the point across that an FPGA can be a secret keeper, without being overly complex. Besides, it was kind of fun to write the obfuscation, and I personally believe that even though the code was pretty simple, the end result is a very difficult system to reverse engineer.

Encoder/Decoder Module

The encoder/decoder module does an obfuscation of the memory by doing simple transformations on the address and data lines. These transformations showcase the strengths and abilities of FPGAs. FPGAs excel at bit manipulations, at implementing digital protocols, and

at calculations of arbitrary bit-widths. These three strengths will be demonstrated in this implementation.

As a side note, for the obfuscation methods presented, a randomization program was written in Java. The program generates compilable Verilog code with sections of the code being randomly generated. The randomized sections create different transformations on the address and data lines. This made it easy for me because I don't have to type as much, and by using the Java encoder/decoder creator program, I can generate several implementations quickly and accurately. In fact, each of the demo boards that we created each have a different obfuscation. Even if you able to decode one board, you only have one badge, and none of the others.

To describe how the obfuscation happens, I will use an English example to demonstrate, using the test phrase "the quick brown fox jumps over the lazy dog."

The first step in obfuscation is a randomized translation of the address lines. This is done through a translation table. Instead of a lookup table for the complete address space, there are smaller lookup tables for groups of 3 address lines, selected at random. The lookup tables are much smaller than the entire address space, so it is quicker to compute, but it is not nearly as random. Also, the top order bits are left alone, allowing for the separation of banks. In our example, the top 5 bits are directly passed through, which gives us 16 different banks. The rest of the bits are arbitrarily split up into groups of 3 wires, selected at random. These 3 wires are then assigned a randomly generated translation table. The result is mangling of the address encoding that is easy to implement, but difficult to decipher without knowing the scrambling key. Below is an example translation table for a single set of 3 wires.

Input Data	Translated Output
000	101
001	001
010	100
011	011
100	110
101	000
110	010
111	111

Figure 8 - Example Translation Table

For our English text example, it would have the effect of turning our test phrase into: "over quick jumps dog fox the the brown lazy". The locations of all of the data are in a randomized location. If you didn't know what the original test phrase is, you might be able to create something like "the lazy dog jumps over the quick brown fox", or "the quick brown dog jumps over the lazy fox", but without a primer, there isn't a reasonable way to re-order the words.

The actual digital effect of the address translation is as follows. Let's say we are going to access bank 1, and each bank is 256k. Our address range would therefore be 0x00040000 : 0x0004FFFF. If the processor wanted to read some bytes from memory, it would present to the FPGA the following addresses and get the following results:

Input Address	Obfuscated Address
23'h040001	23'h07653A
23'h040002	23'h07F139
23'h040003	23'h077538
23'h040010	23'h07E18B
23'h040020	23'h07E10B
23'h040030	23'h07E11B

23'h040100	23’h07E17B
23'h040200	23’h07A13B
23'h040300	23’h07A17B
23'h041000	23’h07E139
23'h042000	23’h06E13F
23'h043000	23’h06E13D

Figure 9 - Example Address Obfuscation

This is an actual example of one of the runs of the obfuscator. Remember that the top 5 bits are unobfuscated. Also, a note on Verilog number format: 23 means number of bits, and the “h” means “represented in hex”, followed by the number. For example, 8’h0F is the 8-bit hex number for 0x0F.

The Verilog code looks like this:

```
case ({AddrIn[17],AddrIn[9],AddrIn[14]})
        3'b000 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b101;
        3'b001 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b001;
        3'b010 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b100;
        3'b011 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b011;
        3'b100 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b110;
        3'b101 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b000;
        3'b110 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b010;
        3'b111 : {AddrOut[17],AddrOut[9],AddrOut[14]} = 3'b111;
endcase

case ({AddrIn[6],AddrIn[3],AddrIn[8]})
        3'b000 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b011;
        3'b001 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b111;
        3'b010 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b010;
        3'b011 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b001;
        3'b100 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b101;
        3'b101 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b100;
        3'b110 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b000;
        3'b111 : {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b110;
endcase

…
```

Figure 10 - Address Obfuscation Verilog code example

For those of you unfamiliar with Verilog, anything in { } means concatenate these bits. So case ({AddrIn[17],AddrIn[9],AddrIn[14]}) means combine those 3 values into a single entity and do a comparison on that. {AddrOut[6],AddrOut[3],AddrOut[8]} = 3'b110 means combine those 3 variables into a single variable and give it an assignment of the binary value “110”.

Note that the three address lines that are selected in the case statement are chosen at random. The conversion table is also generated randomly. Clearly, it would be very difficult to understand the memory contents because the contents are so disorganized. However, the processor doesn’t care, because the FPGA does the conversions. All of the address translation is invisible to the microcontroller. It proceeds as normal without any knowledge that there is a de-obfuscation method happening. The translations are difficult to reverse engineer because the magic is inside the FPGA, where it is hard to retrieve.

The other half of the obfuscation is the data stored at each address. Even if you were able to decode the pattern of the address locations, you would still have to descramble the data. Data scrambling is composed of three parts. The bit order of the data bytes is randomly reordered. Next, a random number of data bits are inverted. Finally, this value is XOR'd with a random selection and ordering of the address. This gives you 8! * 2^8 * (22!-14!)/8! different combinations of scrambling, which is on the order of 3 trillion possibilities. Not bad for 8 bits of data and very simple code. To decode the data, the inverse mechanism is applied, and the data is descrambled.

For the English example, we are now scrambling the letters. "The quick brown fox jumps over the lazy dog" would become: "epC kgHwr ydiUz Niv qgtJl lixcd epC sAbf mIo". (using capital letters to represent inverts). Combined with the address translation, the end result is "lixcd kgHwr qgtJl mIo Niv epC epC ydiUz sAbf". Each data byte is mangled, and the order is randomized.

Surprising to me, the inverse descrambling process required more math than I expected. Let me give an example with 4 bits.

Let's start with a 4-bit nibble [b_3 b_2 b_1 b_0].

Let's set our scrambling mechanism so that it will get to [b_0 b_2 b_3 b_1].

The translation is [3 1 0 2]. (Take the 3rd bit, then the 1st bit, then the 0th bit, then the 2nd bit).

I now need to get back to [b_3 b_2 b_1 b_0]. I would have assumed you do something like use [2 0 1 3] as the inverse matrix. Incorrect.

The real math goes like this:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = [b_0\ b_2\ b_3\ b_1]$$

Then to go backwards, you find the inverse matrix and multiply it by the mangled matrix to get the original order, which will be passed back to the microprocessor:

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_2 \\ b_3 \\ b_1 \end{bmatrix} = [b_3\ b_2\ b_1\ b_0]$$

Who knew! Thank goodness for workmates who graduated from college recently and are familiar with linear algebra.

For the 4-bit example and adding the randomly inserted inversions, we could easily create a transform that looks like this:

[~b_0 ~b_2 b_3 b_1]

The reverse transform would have to be

[b_3 ~b_2 b_1 ~b_0]

Following is an example of the data mangling. It shows the data that enters the FPGA, and then the data that is written to the actual memory location.

Input Data	Obfuscated Data
8'h00	8'h32
8'hFF	8'hCD
8'h01	8'hB2
8'h02	8'h22
8'h04	8'h3A
8'h08	8'h30
8'h10	8'h36
8'h20	8'h72
8'h40	8'h33
8'h80	8'h12
8'hAA	8'h40
8'h99	8'h94
8'h55	8'hBF
8'h66	8'h6B

Figure 11 - Example Data Obfuscation

The Verilog code is pretty simple. An example is shown below:

```
Mem_Din = {MP_Dout[0], MP_Dout[5], ~MP_Dout[7], ~MP_Dout[1], MP_Dout[2],
MP_Dout[4], ~MP_Dout[3], MP_Dout[6]};

MP_Din = (~Mem_Dout[5], Mem_Dout[0], Mem_Dout[6], Mem_Dout[2], ~Mem_Dout[1],
Mem_Dout[3], ~Mem_Dout[4], Mem_Dout[7]};
```

Figure 12 - Example Data Obfuscation Verilog Code

Originally the code didn't have the XOR at the end of it, however it was added to prevent a common reverse engineer technique. The remaining unfilled bytes at the end of a block ended up all being x00's. That might have been enabler for a clever reverse engineer to decode the block. I could have easily just filed it with meaningless noise. But since it was so easy to do, I XOR'd the mangled data with a randomly selected collection of the address bits. That way, even if there are multiple similar bytes "in a row", it would be masked by the different address. Reversing the data to decode it only requires XOR'ing the value with the same patern of address lines. The point here is that there are numerous tools and techniques that the creative engineer could employ with an FPGA to add entropy and obfuscation to the system in order to enable protection of the information. The new code looks something like this:

```
Encoded_Data = mangled_data ^
        {AddrOut[7], AddrOut[12], AddrOut[17], AddrOut[14],
        AddrOut[1], AddrOut[4], AddrOut[8], AddrOut[11]};
```

Figure 13 - Example XOR obfuscation

I also had a little trouble with the math to calculate the number of combinations generated by the XOR gates. What I figured out is that to calculate combinations of randomly selected balls from a bag, the math looks like this:

$$\frac{22}{1} * \frac{21}{2} * \frac{20}{3} * \ldots * \frac{15}{8} = \frac{22!-14!}{8!} = 319770$$

8! * (2 ^ 8) * ((22!-14!)/8!) is 3,300,640,358,400 different encoding possibilities for a byte of data, which is a very impressive number of variance for an 8-bit data source, and not much coding.

I believe that I personally would not want to try to decode the encoded blocks of memory. There is too much entropy in the encoded data from the address translation and the data mangling. Just to reiterate, this is only for data at rest. Once a block is unlocked and is being actively decoded, it is far too easy to watch encoded data enter the FPGA, and watch decoded data exit the FPGA, and from there determine the decoding mechanism. (Or even easier, since the FPGA is set to decode, let it decode the entire block for you.) The simple obfuscation system demonstrates the benefits of protecting information with an FPGA. Having the FPGA as the root of trust adds security by keeping the secret protected information in a place that is practically unreachable.

Conclusion

Using an FPGA is a reasonable and effective technique to protect sensitive information for medium security applications with a limited lifetime. Recovering the circuit in an FPGA configuration bitfile is a difficult process because the information is a convoluted and complex arrangement of proprietary binary data. Using a brute force approach to extract meaning from the bitfile is difficult and time consuming. Using software reverse engineering techniques on the bitfile generation software to extract meaning of the bitfile is an equally daunting, difficult, and time-consuming process. FPGAs protect their content very well. Extracting meaning from them is extremely difficult.

We have then presented a simple example of using the advantages of an FPGA to increase the difficulty to reverse engineer a system that uses an FPGA to protect information. The example uses address translation and data mangling to obfuscate the information stored in an external memory. The process first does an address translation to obfuscate the ordering of the memory contents. The process then does data mangling on each data byte. The end result is an effective method to protect sensitive information.