Chapter 6

REVERSING A LATTICE ECP3 FPGA FOR BITSTREAM PROTECTION

Daniel Celebucki, Scott Graham and Sanjeev Gunawardena

Abstract Field programmable gate arrays are used in nearly every industry, including consumer electronics, automotive, military and aerospace, and the critical infrastructure. The reprogrammability of field programmable gate arrays, their computational power and relatively low price make them a good fit for low-volume applications that cannot justify the non-recurring engineering costs of application-specific integrated circuits. However, field programmable gate arrays have security issues that stem from the fact that their configuration files are not protected in a satisfactory manner. Although major vendors offer some sort of encryption, researchers have demonstrated that the encryption can be overcome. The security problems are a concern because field programmable gate arrays are widely used in industrial control systems across the critical infrastructure. This chapter explores the reverse engineering process of a Lattice Semiconductor ECP3 field programmable gate array configuration file in order to assist infrastructure owners and operators in recognizing and mitigating potential threats.

Keywords: Field programmable gate arrays, threats, reverse engineering

1. Introduction

As field programmable gate arrays (FPGAs) become more powerful and less expensive, they are increasingly being adopted in industry. Key applications areas of FPGAs are industrial control systems used for managing critical infrastructure assets and hardware-in-the-loop simulations used for industrial process system design and training [15]. Low latency, high computational power and an abundance of embedded resources enable FPGAs to implement complex control algorithms with an excellent performance-to-cost ratio. However, FPGAs have security issues that stem from the fact that their configuration files are not protected in a satisfactory manner. A number of attacks targeting FPGAs and FPGA-based systems have been devised. These include hardware Trojans, crippling attacks and fault injection attacks, as well as attacks that reveal sensitive information for subsequent exploitation, such as side-channels, reverse engineering, readback and counterfeiting [2, 4, 9, 16].

This chapter explores the reverse engineering process of a Lattice Semiconductor ECP3 FPGA. The focus is on two key FPGA building blocks – the input/output block and look-up tables. The reverse engineering efforts have resulted in a proof-of-concept parser that analyzes FPGA bitstreams (circuit configuration files) for errors and malicious modifications without revealing any sensitive intellectual property.

2. Background

This section discusses FPGAs, bitstream synthesis, the applications of FPGAs in the critical infrastructure and FPGA threats.

2.1 Field Programmable Gate Arrays

FPGAs were first introduced in 1984 by Xilinx and have since increased in capacity and speed by factors of 10,000 and 100, respectively [17]. Unlike traditional application-specific integrated circuits (ASICs) that are customized for a particular use, FPGAs are reprogrammable. This is accomplished using a combination of configurable logic blocks (CLBs), an input/output block and a series of configurable interconnects. The interconnects are sometimes referred to as the switching matrix.

Figure 1 shows an example FPGA architecture with configurable logic blocks, an input/output block and interconnects. Configurable logic blocks, which comprise digital circuits such as look-up tables, multiplexers and flip-flops, can be configured to perform various combinational functions. These functions can also be registered within a configurable logic block to implement synchronous logic. An input/output block provides connections to external stimuli. The interconnects link the configurable logic blocks and input/output block to complete the desired circuit.

The penalties incurred for FPGA reconfigurability include larger chip area, slower speed and higher power consumption compared with an ASIC that implements the same circuit [6]. This is primarily due to the additional area and propagation delays introduced by the programming circuitry in an FPGA.

The initial steps in designing a digital system are largely identical for FPGAs and ASICs; they involve design capture and simulation using a hardware description language (HDL). After the correct functionality is verified via simulation, the hardware-description-language-based design is synthesized into a form that represents logic elements and registers, which is referred to as the register-transfer level. At this point, the logic elements and registers are mapped to implementable components contained in a target technology library. In the case of ASICs, this is usually a standard cell library. However, for FPGAs, the design is mapped to functional primitives comprising look-up tables and registers. Following the placement and routing, the final design is converted

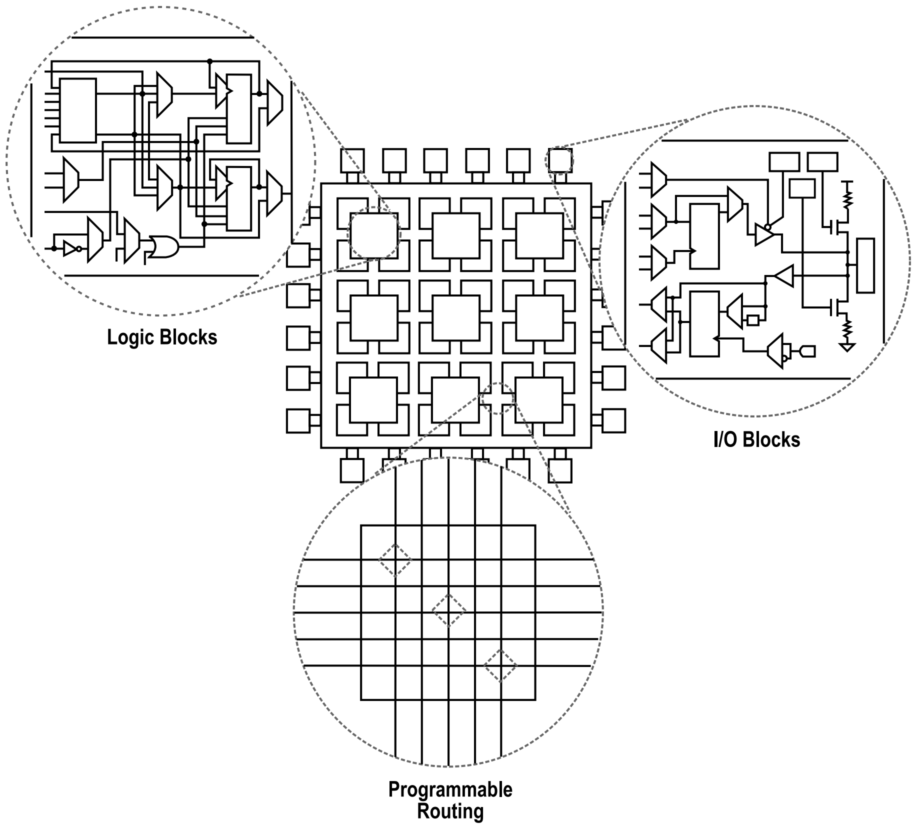

Figure 1. FPGA architecture [12]

.

to a "bitstream," a series of zeros and ones that specifies the configuration options of the configurable logic blocks, input/output block and interconnects in order to implement a given circuit. FPGA vendors have their own proprietary bitstream formats whose details are rarely released to the public.

Configurable Logic Blocks. Configurable logic blocks enable an FPGA to implement logic. Although there are differences in vendor implementations of configurable logic blocks, they commonly include look-up tables, multiplexers and flip-flops. Unlike traditional ASICs that use hardware logic gates to implement digital logic for the desired circuits, FPGAs employ look-up tables. Figure 2 shows a two-input look-up table that uses multiplexers to implement digital logic. Inputs `a` and `b` are selectors for the multiplexers and the inputs to the multiplexers are the output values of the desired truth table. The look-up table implements a circuit that is logically equivalent to an AND gate by setting the inputs to the multiplexers as `0001`. Different input values are provided to

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

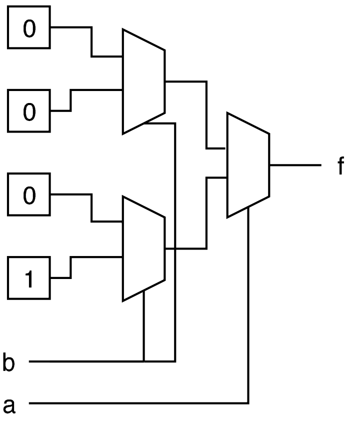

Figure 2. Two-input look-up table example.

the multiplexers to implement a new digital circuit without having to change the hardware. Look-up tables trade space for reprogrammability; the hardware needed to implement a look-up table is larger than that needed to implement the digital circuit replicated by the look-up table. However, a look-up table can be reprogrammed to implement any logic function that can be modeled by a truth table. Designs that require more inputs are implemented by daisy chaining look-up tables.

Input/Output Block. An input/output block connects the internal logic of an FPGA to external components. Since an input/output block usually allows inputs and outputs on the same physical pad, the choice of whether a certain pin is an input or output is decided at configuration time. Other configuration options determine the physical characteristics of the signal at a pin such as pullmode, slew rate and drive level; these may vary from FPGA to FPGA. Figure 3 shows an example input/output block that is configured as an output.

Switching Matrix. A switching matrix connects the configurable logic blocks and the input/output block to produce the desired digital logic circuit [5]. The large number of routes that have to be accommodated make the switching matrix the largest portion of an FPGA in terms of silicon area.

2.2 Bitstream Synthesis

Before a circuit design can be implemented on an FPGA it must be transformed into a configuration file – called a bitstream – that can be loaded on the FPGA. Figure 4 shows how a design proceeds from a hardware description language file to the final bitstream for a specific FPGA. Hardware description language code is first synthesized into a netlist that contains the list of com-

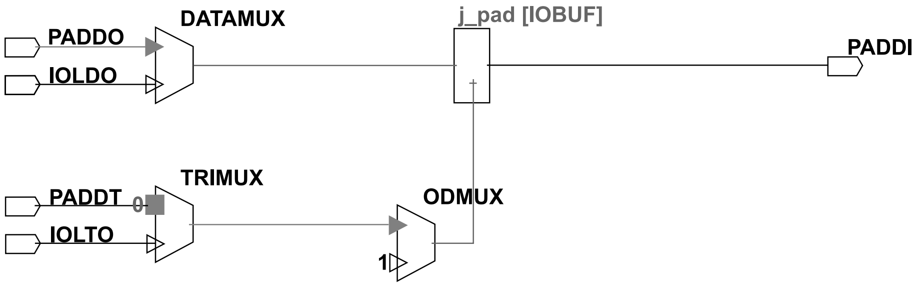

Figure 3. Example input-output block.

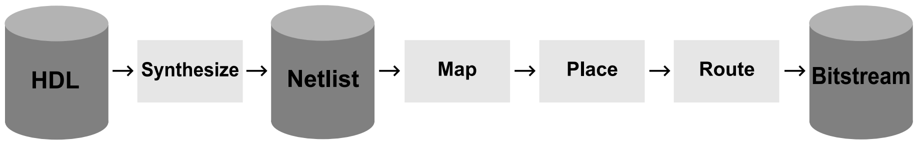

Figure 4. Process for generating a bitstream from HDL [8]
.

ponents in the circuit and the nodes to which they are connected. The map function maps the components in the netlist to the components on the FPGA. The place function then selects the locations of the components on the FPGA. Since the FPGA typically has numerous instances of the same components, the place function determines which components will actually be part of the circuit. The route function then makes the connections between all the placed components on the board. After the circuit has been placed and routed, it is converted to a bitstream file that configures the correct components and connections on the board to create the circuit.

2.3 Critical Infrastructure Applications

Industrial control systems rely heavily on FPGAs. These systems must be powerful and have low latency to ensure high performance, flexibility and reliability [10]. FPGAs are well suited for this purpose. They provide a higher performance-to-cost ratio than ASICs due to continual advances in FPGA computing power and high per-unit costs for low-volume ASIC designs [14]. The reconfigurability of FPGAs supports rapid prototyping as well as control algorithm upgrades throughout the lifespan of an industrial control system using the same deployed hardware. Additionally, system-on-a-chip (SoC) platforms can implement advanced control techniques [15]. Finally, the availability of third-party intellectual property cores that can be licensed or purchased enables infrastructure owners to implement portions of, or complete, FPGA systems by outsourcing the work.

2.4 FPGA Threats

FPGA complexity increases the potential cyber attack surface and, hence, the risk of cyber attacks [1]. These threats can be particularly dangerous to FPGAs used in the critical infrastructure due to the potential impacts on industry, the economy and society.

Bitstream Modification. Chakraborty et al. [2] have demonstrated that a bitstream can be modified to introduce hardware Trojans without knowing the hardware description language (source) code used to create the bitstream. In one instance, they inserted ring oscillators to elevate the temperature of an FPGA, which increased the probability of failure. This attack could be implemented by an insider who is responsible for loading the bitstream on the FPGA or by an adversary who intercepts the original bitstream and delivers the modified bitstream to the FPGA. Although an adversary could simply synthesize a malicious bitstream without ever interacting with the original bitstream, reverse engineering and subsequently modifying the bitstream enable the adversary to implement a hard-to-detect attack that maintains the original functionality of the FPGA design.

Covert Channels. Covert channels allow for the transmission of information between components that are not supposed to be communicating. In an industrial control system setting, this could involve the exfiltration of the control algorithm or sensitive data while the industrial control system is performing its intended functions. The exfiltration of a proprietary control algorithm and sensitive data could give competitors an advantage.

Intellectual Property Theft. An industrial control system vendor that develops its own FPGAs should be wary of adversaries potentially reverse engineering its designs. This is important because bitstreams are not inherently protected and, given enough time, an adversary could reverse engineer them and obtain valuable intellectual property [13]. Malicious entities also would be interested in reverse engineering bitstreams and creating exploits that could be used in future attacks on critical infrastructure assets.

In theory, encryption can be used to protect a bitstream, but this feature is usually offered by expensive FPGA models. In any case, encryption has been shown to be breakable through side-channel analysis [11]. Additionally, encryption requires an energy source, usually in the form of a battery, to keep the key from being cleared if the board loses power. Also, in some cases, an FPGA cannot be accessed after it is deployed or its battery cannot be replaced without significant effort [3].

3. Reverse Engineering Methodology

This section discusses the reverse engineering methodology for a Lattice Semiconductor ECP3 FPGA configuration file.

3.1 Target System

Numerous articles have been published about reverse engineering efforts directed at Xilinx and Altera (now part of Intel) FPGAs. However, Lattice FPGAs have received much less attention apart from the very small iCE40 FPGAs [18]. The target system chosen for this research was the Lattice ECP3 LFE3-35EA-8FN484C FPGA with the Lattice ECP3 Versa Development Kit. Comparisons are made between the reverse engineering process for Lattice FPGAs and the reverse engineering processes for Xilinx and Altera FPGAs.

All the bitstreams were designed using Lattice Diamond software version 3.9.1 and the Lattice Synthesis Engine. Additionally, Tool Command Language scripts were used to generate design variations to explore the effects on the bitstreams.

3.2 Input/Output Block Reversal

This section describes the process of mapping the relationship between a bitstream file and the configuration of the input/output block; this was easily set using the spreadsheet view provided by the Lattice Diamond software. Because changes made to the configuration options in the spreadsheet view were present in the Lattice preference file (LPF) when the changes were saved, modifying the Lattice preference file directly enabled the automated generation of a bitstream.

The reverse engineering of the input/output block has three goals: (i) map a number of the configuration options for each pin to their respective indices and values in the bitstream file; (ii) determine whether a pin is an input or output based on the bitstream file; and (iii) determine whether a pin is connected to logic blocks in the design based on the bitstream file.

Pullmode. Although input/output blocks have a variety of configuration options, the reverse engineering process of the different configuration options is very similar. Therefore, only the process for reverse engineering the pullmode attribute is described here.

The pullmode is responsible for describing how a signal is interpreted at a pin. The pullmode can be set to the following four modes:

- **Up:** The input is attached to a pull-up resistor, i.e., the pin is tied to a logical 1.
- **Down:** The input is attached to a pull-down resistor, i.e., the pin is tied to ground or a logical 0.
- **Keeper:** This mode is neither pull-up nor pull-down. It drives a weak 0 or 1 level to match the level of the last logic state present on the pad to prevent the pad from floating.
- **None:** The input is not set to any of the above three modes.

Algorithm 1 : Pullmode bitstream generation.

```
1: for Pin p in all I/O Pins do
2:     for val in UP, DOWN, KEEPER, NONE do
3:         Replace IOBUF line in LPF with "IOBUF PORT "a" PULLMODE=val"
4:     end for
5:     Replace Location line in LPF with "LOCATE COMP "<input/output pin
       name>" SITE p"
6: end for
```

In this chapter, an index refers to the byte address where the contents of a bitstream have been changed due to a design modification. A change to the pullmode of a pin resulted in three to six change indices in the bitstream. This was much more manageable compared with the 70 to 100 indices when the location of a configurable logic block was moved slightly. The reason for the varying change indices is because the bitstream did not abide by the byte boundaries; this is discussed later in this chapter.

After the configuration option was sufficiently isolated, a Tool Command Language script synthesized bitstreams for every pullmode option for every pin; every pin set was first used as an input and subsequently every pin set was used as an output. Algorithm 1 shows the pseudocode of the script. The objectives were to determine which indices were responsible for the pullmode option for each pin and whether a common pattern could be used for every pin to identify the pullmode. The hypothesis was that each pin would have a different location in the bitstream where its configuration options were stored, and the values at each location would follow the same pattern in terms of representing the pullmode in the bitstream.

The 2,296 bitstreams generated by the script were compared to find the indices responsible for the pullmode configuration option for each pin. Table 1 shows the bitstream indices responsible for the pullmode configuration option for six pins. The indices for each pin were generated by comparing the four bitstreams (pull-up, pull-down, bus keeper and none) for the pin and listing all the indices in the bitstreams that were different from any of the other bitstreams. For each pin, the first column with hex values (i.e., second column overall) refers to the values in the bitstream associated with pullmode pull-up, the second refers to pull-down, the third refers to bus keeper and the fourth column refers to none. The numbers in the leftmost (i.e., first) column are the bitstream indices where the changes occurred. For example, when comparing the four bitstreams generated for pin A2 and set as an input, the only differences between the four bitstreams were at bytes 429, 476 and 477. In fact, Table 1 reveals that relatively few indices were changed.

Note that indices 476 and 477 appear for almost every pin. However, for the generated bitstreams, it was impossible to know whether all the indices listed for each pin were necessary to configure the various pullmode options or if only a subset of the indices for each pin was necessary.

Table 1. Indices for the pullmode configuration option for various pins.

Pin A2					**Pin A3**				
429	00	06	02	04	416	00	06	02	04
476	f3	9e	57	3a	476	20	5e	0a	74
477	dc	b4	07	6f					
Pin A4					**Pin A6**				
412	00	18	08	10	395	01	61	21	41
476	82	e2	a2	c2	476	d4	cd	5c	45
477	ba	ea	8a	da	477	bf	30	39	b6
Pin A7					**Pin A8**				
326	01	61	21	41	308	00	01	00	01
476	c5	63	27	81	309	04	84	84	04
477	02	ab	66	cf	476	47	e3	a4	00
					477	7c	44	97	af

To solve this problem the bitstream generation script was executed again with different logic designs mapped to different portions of the FPGA. The reasoning was to isolate the indices responsible for the pullmode configuration option. If the same generation script was executed with different logic designs and the logic mapped to different portions of the FPGA, then the indices responsible for the pullmode configuration would have the same values across all the runs while the indices affected by the switching matrix or configurable logic blocks would change. The following gates and placements were employed:

- One-input NOT gate at R2C73D.
- One-input NOT gate at R23C53A.
- Two-input AND gate at R3C70B.
- 1553 encoder placed by the compiler.

When exploring the designs and placements required to isolate the indices, it became clear that the configurable logic blocks had to be varied in diverse ways. This was achieved using a NOT gate, an AND gate and the intellectual property core of a MIL-STD-1553 encoder. The simple gates represented small designs whereas the encoder represented a large design. Each gate was then placed in a different slice within the configurable logic blocks on different corners of the FPGA and the 1553 encoder was placed by the tool. This variation proved to be enough initially. If none of the indices expressed different values, then additional variation could be introduced before considering that all the indices listed were necessary to represent the pullmode configuration.

The assumption that all the indices listed were responsible for the pullmode was excluded for a few reasons. First, the number of indices that changed for each pin when comparing bitstreams was not constant. Some pins only had three indices change whereas other pins had six indices change. It is unlikely that a designer would use extra indices to represent the same change in different pins. Additionally, each pin had indices that were different, but some pins also had indices that were the same. A designer would likely not have the same information located in two locations, especially since a larger file would increase the FPGA configuration time and complexity of the process used to parse the bitstream. In fact, it is more likely that some indices were being changed due to some other variation that was occurring as a result of changing the pullmode. The difference in the numbers of changed indices and shared indices was later attributed to the bitstream not abiding by the byte boundaries and some indices acting as internal checksums.

After analyzing which indices were changing for each pin, the pins were organized into six groups based on the indices responsible for their changes. Pins that shared the same indices were grouped together as well as pins that shared a pattern in the offset of their indices. The binary values located at all the indices in each of the six groups were then printed for each pullmode for each pin in a group, facilitating the visual inspection of all the indices simultaneously in order to discern changes. If the hypothesis was correct, there would be a regular pattern of 1s and 0s cascading down the created file. To facilitate visual inspection, 1s were replaced with black spaces and 0s with white spaces.

Figure 5 shows a selection of the pins in the first group after the 1s were replaced with black spaces and 0s with white spaces. A single bitstream runs horizontally from left to right and each group of four bitstreams relates to the same pin. For example, the first four bitstreams in Figure 5 are the bitstreams related to the pullmode configuration of D19. The first is pull-up, the second pull-down, the third bus-keeper and the fourth none. The next four bitstreams follow the same pullmode pattern, but for pin D18, the next four for pin B20, and so on.

Additionally, the first bitstream for each pin is always pullmode up, followed by down, keeper and none. The figure reveals significant information about how the bitstream is organized with respect to the pins. First, the bitstreams do not adhere to strict byte boundaries. The columns of black squares running vertically through the picture represent 1s that are the boundaries for where information about a certain pin appears. The 1s between the columns correspond to different configuration options. The pullmode configuration option can be observed at each of the pins as the only change in a pin's space in the boxes. Pullmode up is represented as `00`, down as `11`, keeper as `01` and none as `10`. This pattern was observed in all six pin groups investigated in the research.

3.3 Configurable Logic Block Reversal

Configurable logic blocks were more difficult to reverse engineer than the input/output block because there are many more configurable logic blocks, and the Lattice preference file modification cannot be used to change the configuration options in a straightforward manner. This is because the look-up tables in the configurable logic blocks are configured based on the hardware description

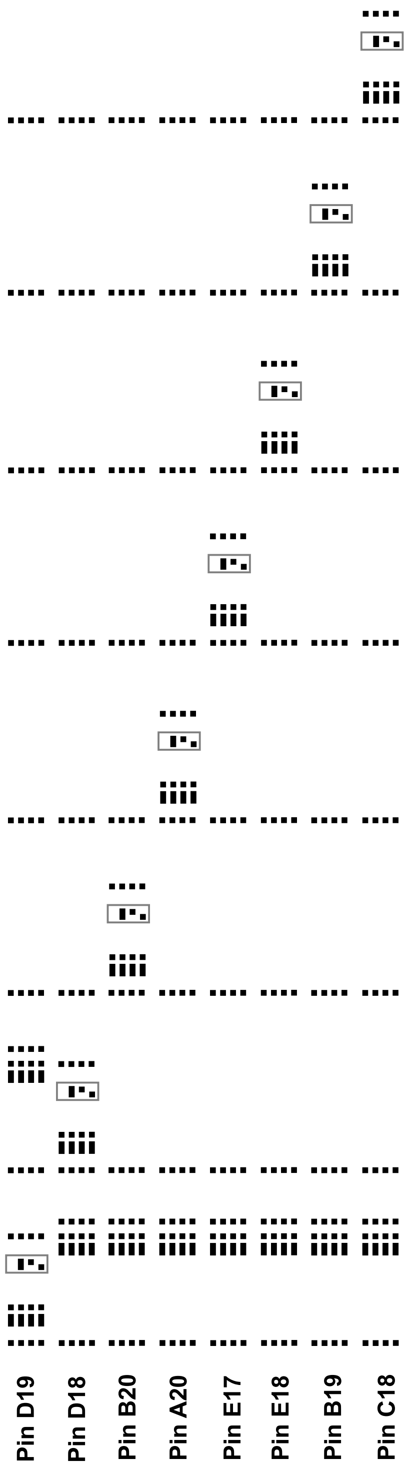

Figure 5. Selection of bitstreams.

Table 2. Derived truth table.

A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
B	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
C	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
D	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F	0	0	1	0	0	0	1	0	0	0	1	0	1	1	1	1

language when they were synthesized. The Lattice preference file is not used until the map, place and route steps in the bitstream generation process; therefore, it was not even considered until after the look-up tables were configured.

In order to overcome this issue, Lattice primitives were used along with hardware description language attributes. Each Lattice FPGA has a library of primitives supported by the device. In the case of the ECP3, the LUT4 primitive was used so that the look-up table could be directly initialized to the desired configuration value. The hardware description language attributes were used to set other attributes such as location instead of modifying the Lattice preference file simply to avoid having to change two different files. The look-up tables were initialized based on the outputs of their desired truth tables shown in Table 2. An initialization value of `0xF444` yielded a look-up table that produced the outputs in the truth table. When the synthesis process translates hardware description language code to the bitstream, it replaces the logic in the design with the look-up tables that are initialized to produce the same outputs. Based on this information, it was hypothesized that the initialization information appears somewhere in the bitstream. Therefore, in order to understand the digital logic implemented in the configurable logic block, it is only necessary to determine how the look-up tables were initialized and then recreate the truth table.

Single Look-Up Table Reversal. The process for reverse engineering the configuration of a look-up table involved the creation of a set of bitstreams using Tool Command Language scripts that had a variety of configuration values for the same look-up table. The bitstreams were compared to identify the indices that were responsible for the configuration information. The bitstreams were then visually compared with each other at the indices to reveal how the configuration information was encoded in the bitstream.

Since each look-up table has a 16-bit configuration value, the 16-bit value was assumed to be stored somewhere in the bitstream. Therefore, at least sixteen bitstreams had to be generated for each look-up table in order to locate the indices. However, if other indices were also changed as a result of modifying the configuration value, additional bitstreams may be necessary to identify the correct configuration indices. Therefore, in the experiments, 61 bitstreams were initially generated for the look-up table – `0x0` through `0xF` for each symbol in

```
0000  0000111111110000  0010100111100110  0000111111110000  0010110110111101
0001  0000111111110000  0010100111100110  0000111111100000  1111011111110011    1's place
0002  0000111111110000  0010100111100110  0000111111010000  0001100100100100    2's place
0003  0000111111110000  0010100111100110  0000111111000000  1100001101101010
0004  0000111111110000  0010100111100110  0000111011110000  0000100101110000    4's place
0005  0000111111110000  0010100111100110  0000111011100000  1101001100111110
0006  0000111111110000  0010100111100110  0000111011010000  0011110111101001
0007  0000111111110000  0010100111100110  0000111011000000  1110011110100111
0008  0000111111110000  0010100111100110  0000110111110000  0110010000100111    8's place
0009  0000111111110000  0010100111100110  0000110111100000  1011111001101001
000A  0000111111110000  0010100111100110  0000110111010000  0101000010111110
000B  0000111111110000  0010100111100110  0000110111000000  1000101011110000
000C  0000111111110000  0010100111100110  0000110011110000  0100000011101010
000D  0000111111110000  0010100111100110  0000110011100000  1001101010100100
000E  0000111111110000  0010100111100110  0000110011010000  0111010001110011
000F  0000111111110000  0010100111100110  0000110011000000  1010111000111101
```

Figure 6. Bitstreams with different configuration values.

the four-digit hex number. This provided adequate information to determine how the configuration was stored in the bitstream.

When comparing a set of bitstreams with different configuration options for the same look-up table, between six to eight bytes were observed to change in the bitstream. The variation in the number of changed bytes has to do with the bitstream not abiding by the byte boundaries and the presence of some checksum-like bits that also change. Sixteen indices correspond to the 16-bit initialization value used for look-up table configuration and 32 bits serve as a checksum. However, the configuration information is encoded. If the initialization value of the look-up table is considered to a 16-bit binary number, then the indices are negated in that 1s are replaced with 0s, and vice versa. Additionally, the 16-bits are not placed next to each other, but are spread across two to four bytes that can be hundreds of indices apart in the bitstream. The bits responsible for encoding the configuration information were discerned by analyzing the differences between the individual bitstreams.

Figure 6 illustrates this process. Each line corresponds to one of the first sixteen bitstreams from look-up table 1 in R2C40D, each with a different configuration value. The four values on the left show the hex representation of the 16-bit value used to initialize the look-up table and the indices enclosed by boxes correspond to the 1-, 2-, 4- and 8-place locations of the corresponding hex value. This is observed by comparing which locations change from line to line. For example, the 1-place for the first hex value was confirmed by comparing the `0x0002` and `0x0003` initialization value lines. All the indices in the last sixteen indices were not changed in a regular manner, so they can be ignored. However, the change from `0x0002` to `0x0003` has a 0 in in the same location where the `0x0001` line has a 0. This was also confirmed by comparing the `0x0004` line and the `0x0005` line or any other line where the binary representation was changed in the 1-place. This process was repeated for the remaining configuration values.

After the process was completed, many of the remaining bitstreams were removed to reveal the mask shown in Figure 7. The mask is the set of locations

```
0001 00001111 11110000 00101001 11100110 00001111 11100000 11110111 11110011  1 place
0002 00001111 11110000 00101001 11100110 00001111 11010000 00011001 00100100  2 place
0004 00001111 11110000 00101001 11100110 00001110 11110000 00001001 01110000  4 place
0008 00001111 11110000 00101001 11100110 00001101 11110000 01100100 00100111  8 place
0010 00001111 11110000 00101001 11100110 00001111 10110000 01000100 10001111  16 place
0020 00001111 11110000 00101001 11100110 00001111 01110000 11111111 11011001  32 place
0040 00001111 11110000 00101001 11100110 00001011 11110000 10111110 10001001  64 place
0080 00001111 11110000 00101001 11100110 00000111 11110000 10001011 11010000  128 place
0100 00001111 11100000 11110011 10101000 00001111 11110000 00101101 10111101  256 place
0200 00001111 11010000 00011101 01111111 00001111 11110000 00101101 10111101  512 place
0400 00001110 11110000 00001101 00101011 00001111 11110000 00101101 10111101  1024 place
0800 00001101 11110000 01100000 01111100 00001111 11110000 00101101 10111101  2048 place
1000 00001111 10110000 01000000 11010100 00001111 11110000 00101101 10111101  4096 place
2000 00001111 01110000 11111011 10000010 00001111 11110000 00101101 10111101  8192 place
4000 00001011 11110000 10111010 11010010 00001111 11110000 00101101 10111101  16384 place
8000 00000111 11110000 10001111 10001011 00001111 11110000 00101101 10111101  32768 place
```

Figure 7. Mask for R2C40D look-up table 1.

that encode the 16-bit initialization value for the look-up table. The same process was then performed for the remaining look-up tables on the board to obtain their masks.

Table 3. HDL used to generate a three-input AND gate.

```
module four_and_gate (a, b, c, d, i) /* synthesis LOC="R2C40D" */;
   input a /* synthesis LOC="E18" */;
   input b /* synthesis LOC="B20" */;
   input c /* synthesis LOC="A20" */;
   input d /* synthesis LOC="D18" */;
   output i /* synthesis LOC="D19" */;
assign i = a&b&c;
endmodule
```

Mask Correctness Confirmation. After the mask for a specific look-up table was fully reversed, it was necessary to confirm that the information was correct and useful for understanding a bitstream. To accomplish this, bitstreams for a three-input AND gate and a more complicated design of $AB + C\bar{D}$ were synthesized at look-up table 1 in the R2C40D configurable logic block. As shown in Tables 3 and 4, both were designed using hardware description language operators instead of initializing the primitives directly to ensure the initialization value in the bitstream was generated by the synthesis engine when translating the design. The bitstreams were then inspected at the locations corresponding to the look-up table configuration value and the truth tables were recovered.

Table 5 shows the bitstream values (underlined) at the R2C40D look-up table 1 indices for a three-input AND gate. Compared with the mask for the look-up table shown in Figure 7, there are 0s in the 16,384-place and 32,768-place, resulting in a hex value of `0xC000`. When this value was used to derive

Table 4. HDL used to generate a more complicated logic function.

```
module four_logic_gate (a, b, c, d, i) /* synthesis LOC="R2C40D" */;
   input a /* synthesis LOC="E18" */;
   input b /* synthesis LOC="B20" */;
   input c /* synthesis LOC="A20" */;
   input d /* synthesis LOC="D18" */;
   output i /* synthesis LOC="D19" */;
assign i = (a&b)|(c&~d);
endmodule
```

Table 5. Bitstream values for a three-input AND gate.

```
00000011 11110000 00011100 10111111 00001111 11110000 00101101 10111101
```

the truth table, the inputs `1110` and `1111` yielded an output of `1`. This matches the logic for a three-input AND gate, implying that the mask is correct.

Table 6. Bitstream values for a more complicated logic function.

```
00000100 11000000 00001100 00001011 00000101 11110000 11000010 01001010
```

The logic for the second design is more difficult to reconstruct. Table 6 shows the bitstream values (underlined) at the R2C40D look-up table 1 indices.

Table 7 shows the reconstructed truth table. This truth table was used to recover the digital logic function:

$$\bar{A}\bar{B}CD + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}CD + ABCD$$

which can be further reduced to:

$$A\bar{B} + CD$$

Although this function is not in the same form as the hardware description language, it is important to note that the synthesis process has control over how the inputs are routed to the look-up table. This is logically equivalent to what was specified in the hardware description language and that the mask can be used to correctly predict the logic in a look-up table. Although the routing cannot be inferred, it is still possible to understand the logic function embodied in a look-up table by analyzing the bitstream. Thus, the logic embodied in every look-up table on the board can be analyzed although the connections between the look-up tables are not fully reverse engineered.

Table 7. Recovered truth table.

W	X	Y	Z	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

3.4 Bitstream Modification Attack

A bitstream modification attack was attempted using the information gained via reverse engineering – specifically, how the configuration information for a look-up table was stored. The goal was to simulate an attack where an adversary intercepts a bitstream *en route* from the designer to the target system. The adversary then modifies the bitstream, which is loaded on the target system. This demonstrates the feasibility of a more complicated attack than the hardware Trojan described in [2] and further confirms the validity of the look-up table mask.

Experimental Design. The initial logic function chosen was a simple four-input OR gate. The OR gate was implemented using hardware description language operators instead of configuring the look-up table directly. This ensured that the attack scenario would be similar to the actual process involving an intellectual property design. The inputs were connected to four dip switches based on the Lattice preference file constraints and the output was connected to an LED. After confirming that the design was functioning correctly on the target system, the bitstream was modified directly to implement a four-input AND gate and the new design was loaded on the target system, where the functionality was observed.

Modification Results. Table 8 shows the hardware description language code used to generate the four-input OR gate. Table 9 shows the Lattice preference file used to incorporate the design in a look-up table. The design

Table 8. HDL used to generate the OR gate for bitstream modification.

```
module orgate (a,b,c,d,e);
   input a,b,c,d;
   output e;
assign e = a|b|c|d;
endmodule
```

Table 9. LPF constraints used to generate the OR gate for bitstream modification.

```
BLOCK RESETPATHS;
BLOCK ASYNCPATHS;
Locate comp "a" site "j7";
Locate comp "b" site "j6";
Locate comp "c" site "h2";
Locate comp "d" site "h3";
Locate comp "e" site "u19";
Locate comp "orgate" site "r2c40d";
```

was then loaded on the board and the correct functionality was observed. On this board, the LEDs turned off when they were driven high.

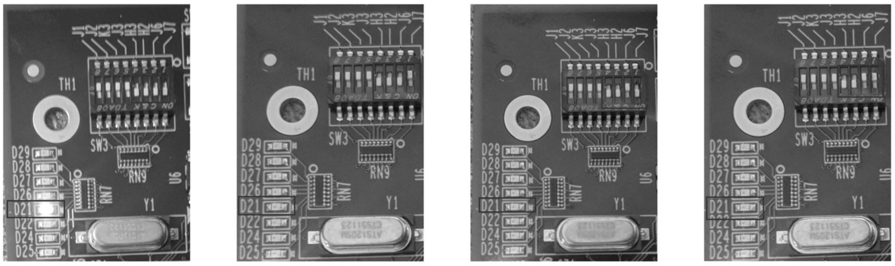

Figure 8. Dip switch states showing the correct function of an OR gate.

Figure 8 shows a subset of the states for the OR gate, demonstrating that the LED correctly lit up when the input was 0000 and the LED was turned off for all other inputs. The bitstream was then modified at the indices related to the first look-up table in the R2C40D configurable logic block.

Table 10 shows the indices that were replaced in the bitstream. The underlined indices correspond to the look-up table configuration bits and the indices in bold font correspond to the checksum bits. The checksum bits were identified when attempting to load the modified bitstream on the target system. The programmer tool was able to detect the modifications and returned an invalid file report or an XCF file reading error. (An XCF configuration file contains information about the device, data files targeted and the operations

Table 10. Indices modified to transform an OR gate into an AND gate.

OR Gate			
00000000	00000000	**00100100**	**01101001**
00000000	00010000	**11111010**	**01111100**
AND Gate			
00000111	11110000	**10001111**	**10001011**
00001111	11110000	**00101101**	**10111101**

to be performed [7].) However, when the checksum bits were replaced with the checksum bits obtained by reverse engineering the mask, the programmer tool accepted the file.

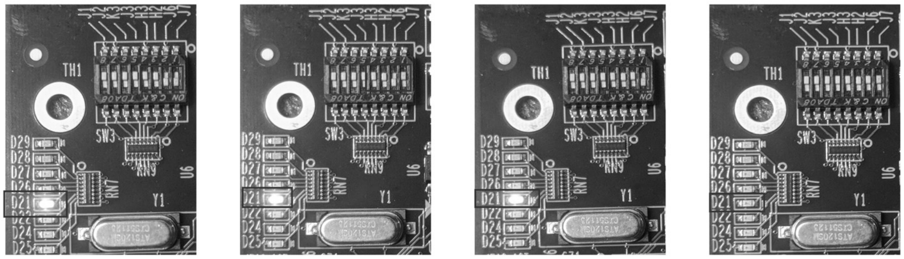

Figure 9. Dip switch states showing the correct AND gate function after the attack.

Figure 9 shows the target system after the modified bitstream attack. The LED was turned on for every input except for `1111`. This demonstrates that the correct behavior was obtained after loading the modified bitstream on the target system, implying that the attack was successful. Although the presence of the checksum bits increased the difficulty of the modification attack and the use of pre-synthesized checksums was not feasible, the checksum can, in fact, be defeated. In the case of a simple modification, the checksum can be brute-forced because the indices that are verified by the checksum are known.

The other option is to reverse engineer the checksum algorithm. This is accomplished by running the programmer or the Lattice Diamond software through a debugger to observe the operations that compute and verify the checksum. This method has been used in a similar scenario where the encryption schemes used by the Stratix II and Stratix III FPGAs were defeated [16].

4. Experimental Results

The experiments demonstrate that the locations and values of the various configuration options of the Lattice ECP3 LFE3-35EA-8FN484C FPGA could be reverse engineered successfully. In the case of the input/output block, the pullmode locations and values were reverse engineered for every pin. For the slew rate, drive level, input and output configuration options, the locations and values were found for all the pins in Groups 1 through 4, as well as one pin in Group 5. This equates to a total of 139 pins. For the configurable logic blocks, the encoding of the configuration information for a small set of the look-up tables was located and recorded, which was used in the successful bitstream modification attack.

This research also created a bitstream parser. The parser processes a bitstream synthesized for the LFE3-35EA-8FN484C FPGA and outputs configuration information about the reverse-engineered input/output block. The information obtained for each look-up table after it was fully reversed is passed to the parser to obtain additional information such as the percentage of the look-up table utilized and its logic function.

Although the bitstream parser does not provide complete information about a bitstream, it should be of value to the industrial control system community. Consider a scenario where a critical infrastructure asset owner receives an updated bitstream from a vendor. Running the new and old bitstreams through the parser would help detect errors and/or malicious modifications. The addition of new input or output pins would indicate potential covert channels. Large increases in look-up table utilization could indicate the insertion of malicious hardware. Although the bitstream parser was developed specifically for the LFE3-35EA-8FN484C FPGA, the underlying process can be applied to other Lattice FPGAs that use the Lattice Diamond software, and, with some modifications and enhancements, to other FPGAs.

5. Conclusions

FPGAs are commonly used in critical infrastructure assets. Their power-to-cost ratio and their reprogrammability make them particularly attractive for industrial control applications. However, their complexity increases the risk of attacks. This chapter has demonstrated the process of reverse engineering a portion of a previously-unexplored Lattice FPGA, which has been incorporated in a parser that enables the analysis of bitstreams for errors and malicious modifications without revealing any sensitive intellectual property.

Future research will continue the reverse engineering efforts on the switching matrix and also concentrate on other FPGAs. Additionally, research will focus on automating the reverse engineering process for Lattice FPGAs and FPGAs from other vendors.

Note that the views expressed in this chapter are those of the authors and do not reflect the official policy or position of the U.S. Air Force, U.S. Department of Defense or U.S. Government.

References

[1] J. Brenner, Keeping America Safe: Toward More Secure Networks for Critical Sectors, MIT Center for International Studies, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2017.

[2] R. Chakraborty, I. Saha, A Palchaudhuri and G. Naik, Hardware Trojan insertion by direct modification of FPGA configuration bitstream, *IEEE Design and Test*, vol. 30(2), pp. 45–54, 2013.

[3] Z. Ding, Q. Wu, Y. Zhang and L. Zhu, Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation, *Microprocessors and Microsystems*, vol. 37(3), pp. 299–312, 2013.

[4] S. Drimer, Security for Volatile FPGAs, Technical Report UCAM-CL-TR-763, Computer Laboratory, University of Cambridge, Cambridge, United Kingdom, 2009.

[5] U. Farooq, Z. Marrakchi and H. Mehrez, Chapter 2, FPGA architectures: An overview, in *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*, Springer, New York, pp. 7–48, 2012.

[6] I. Kuon and J. Rose, Measuring the gap between FPGAs and ASICs, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26(2), pp. 203–215, 2007.

[7] Lattice Semiconductor, Lattice Diamond 3.4 Help, Hillsboro, Oregon, 2014.

[8] E. Lubbers, Configurable System-on-Chip: Xilinx EDK, University of Paderborn, Paderborn, Germany (`slideplayer.com/slide/5083550`), 2014.

[9] S. Mal-Sarkar, A. Krishna, A. Ghosh and S. Bhunia, Hardware Trojan attacks in FPGA devices: Threat analysis and effective countermeasures, *Proceedings of the Twenty-Fourth Edition of the Great Lakes Symposium on VLSI*, pp. 287–292, 2014.

[10] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan and M. Naouar, FPGAs in industrial control applications, *IEEE Transactions on Industrial Informatics*, vol. 7(2), pp. 224–243, 2011.

[11] A. Moradi, A. Barenghi, T. Kasper and C. Paar, On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs, *Proceedings of the Eighteenth ACM Conference on Computer and Communications Security*, pp. 111–124, 2011.

[12] National Instruments, Introduction to FPGA Hardware Concepts (FPGA Module), Austin, Texas, 2011.

[13] J. Note and E. Rannaud, From the bitstream to the netlist, *Proceedings of the Sixteenth International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pp. 264–272, 2008.

[14] J. Rodriguez-Andina, M. Moure and M. Valdes, Features, design tools and application domains of FPGAs, *IEEE Transactions on Industrial Electronics*, vol. 54(4), pp. 1810–1823, 2007.

[15] J. Rodriguez-Andina, M. Valdes-Pena and M. Moure, Advanced features and industrial applications of FPGAs – A review, *IEEE Transactions on Industrial Informatics*, vol. 11(4), pp. 853–864, 2015.

[16] P. Swierczynski, A. Moradi, D. Oswald and C. Paar, Physical security evaluation of the bitstream encryption mechanism of Altera Stratix II and Stratix III FPGAs, *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7(4), article no. 34, 2015.

[17] S. Trimberger, Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology, *Proceedings of the IEEE*, vol. 103(3), pp. 318–331, 2015.

[18] C. Wolf, Project IceStorm (`www.clifford.at/icestorm`), 2018.