University of Cincinnati

Date: 6/28/2022

I, Heiko Stowasser, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:

An Abstract Approach To FPGA LUT Bitstream Reverse Engineering

Student's name: **Heiko Stowasser**

This work and its defense approved by:

Committee chair: John Emmert, Ph.D.

Committee member: Carla Purdy, Ph.D.

Committee member: Ranganadha Vemuri, Ph.D.

43032

An Abstract Approach To FPGA LUT Bitstream Reverse Engineering

A thesis submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Department of Electrical

Engineering and Computer Science

by

Heiko Stowasser

B.S. University of Cincinnati

May 2020

Committee Chair: John Emmert, Ph.D

June 28, 2022

Abstract

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed so they can be reprogrammed to implement any logic circuit. FPGAs are used in critical systems like military radar and wireless communication infrastructure, making FPGA security critical. One major threat to the security of FPGAs are Trojans. Trojans are malicious modifications made to a circuit at any point in the design process. The reprogrammable nature of FPGAs makes them doubly vulnerable to Trojans because even if the physical chip is secure Trojans can still be inserted by compromising the bitstream that programs the FPGA. These types of Trojans could be detected by analyzing the bitstreams of affected FPGAs. However, FPGA manufacturers do not publish the format of bitstreams, providing a layer of inherent obfuscation for attackers to exploit. Meaning that the format of an FPGA's bitstream must be reverse engineered before it is possible to analyze the bitstream for Trojans. Existing methodologies for reverse engineering FPGA bitstreams require expert knowledge of an FPGA's architecture and its associated toolchain. In this Thesis we demonstrate a methodology of reverse engineering FPGA Look-Up-Tables (LUTs), the fundamental component of FPGA reprogrammable logic. Our methodology uses generic VHDL, which allows it to be easily ported to different FPGAs with only basic knowledge of FPGA design flow.

Acknowledgements

I want to thank my advisor, Dr. John Emmert, for guiding me through this project. I would also like to thank Dr. Ranga Vemuri and Dr. Carla Purdy for serving as members of my committee. The Hardware Security course taught in part by Dr. Emmert and Dr. Vemuri, and the Embedded Systems Security course taught by Dr. Purdy enlightened me to the challenges posed in the field of Hardware Security and had a profound effect on my academic path. Prior to me taking these courses in my final year as an undergrad I had no interest in perusing a Master's degree. Those courses sparked and interest in Hardware Security that I will carry forward into my career.

I also want to thank my lab mate, Anvesh, for sharing his knowledge of FPGAs. His work ethic was inspiring and his interest in the field of Hardware Security was contagious.

I am thankful to my family and friends for their support. Especially to Linden, for encouraging me every step of the way.

Finally, thank you to NSF Center for Hardware and Embedded Systems Security and Trust (CHEST) IUCR for sponsoring this research.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 What are FPGAs . . . 1

1.1.2 What are Trojans . . . 2

1.1.3 Reverse Engineering FPGA bitstreams . . . 2

1.1.4 Why reverse engineer FPGA bitstreams . . . 3

1.1.5 How have FPGAs been REd in the past . . . 3

1.1.6 What does it mean to RE FPGAs abstractly . . . 3

1.1.7 What are the benefits of an abstract approach . . . 4

1.2 Outline . . . 4

2 Background **5**

2.1 FPGA Background . . . 5

2.1.1 FGPA Architecture . . . 5

2.1.2 Logic Blocks . . . 6

2.1.3 Look-Up-Tables . . . 7

2.1.4 LUT Decomposition . . . 8

2.1.5 FPGA design flow . . . 9

2.2 Prior Work . . . 9

2.2.1 Project X-Ray . . . 9

2.2.2 Logarithmic-Time FPGA Bitstream Analysis . . . 10

2.3 Summary . . . 11

3 Methodology **12**
3.1 Generic LUT Mapping . . . 12
3.1.1 Challenges . . . 12
3.1.2 Method . . . 14
3.1.3 LUT Interconnect Patterns . . . 15
3.1.4 Solving the LUTs . . . 19
3.1.5 Sorting LUT bits . . . 25

4 Results **28**
4.1 Test Methodology . . . 28
4.2 LUT Utilization Results . . . 28
4.3 Problems . . . 29
4.4 LUT Solving Results . . . 31
4.5 Summary . . . 34

5 Conclusion **35**
5.1 Future Work . . . 35
5.2 Closing Thoughts . . . 36

A **37**
A.1 Acronyms . . . 37

B **38**
B.1 Quartus . . . 38
B.1.1 Quartus Build Script . . . 38
B.2 Vivado . . . 39
B.2.1 Vivado Build Script . . . 39

List of Figures

2.1 FPGA Island Style Architecture 6
2.2 Example Logic Block from 7 Series FPGAs CLB User Guide UG474 [10] . . 7
2.3 Three Input LUT 8
2.4 FPGA Design Flow 9

3.1 Input Pin Reordering 13
3.2 Row/Column Method 17
3.3 Snake Methods 19
3.4 Mask Generation 21
3.5 Pseudo-Binary Search 23
3.6 LUT Bit Sorting 27

List of Tables

3.1 LUT Bit Hamming Weight 14

4.1 Maximum Single Run LUT Utilization Achieved 29

4.2 LUT Solving Data 31

4.3 Row/Col Method: Run 1 32

4.4 Row/Col: Run 2 32

4.5 Snake: Run 1 32

4.6 Snake: Run 2 33

4.7 Modified Snake: Run 1 34

4.8 Modified Snake: Run 2 34

Chapter 1

Introduction

1.1 Motivation

1.1.1 What are FPGAs

FPGAs or Field Programmable Gate Arrays are integrated circuits designed so that the logic they implement can be reprogrammed. FPGAs contain an array of logic blocks connected by a reconfigurable grid of interconnects. The structure of these logic blocks are detailed further in Chapter 2, but they are elements that can be programmed to implement arbitrary logic circuits. The circuit that the FPGA implements is configured by a programming file. The programming files used to configure FPGAs are commonly referred to as bitstreams. These bitstreams are exactly as the name suggests, an array of ones and zeros that contain the information to configure the FPGA to the desired circuit. The reprogrammable nature of FPGAs makes them useful for applications that cannot be done efficiently by a general-purpose processor, and do not have a high enough volume of end products to justify the high upfront cost of developing an Application Specific Integrated Circuit (ASIC). FPGAs have a significantly shorter time-to-market than traditional ASICs and they also offer more flexibility than a traditional ASIC, with the tradeoff of usually having a higher per-unit cost.

1.1.2 What are Trojans

Trojans are malicious modifications made to a circuit at any point in the design process. There are many different types of Trojans, some are intended as backdoors to leak secret information or gain control over a system while others serve as kill switches to allow attackers to disable or destroy a device. Trojans pose a major threat to the security of FPGAs because Hardware Trojans can be physically added to an FPGA in all the same ways as ASICs, anywhere along the design or fabrication steps. Moreover, FPGAs are reprogrammable. This makes them doubly vulnerable to Trojans because even if the physical chip is secure Trojans can still be inserted by compromising the bitstream. By tampering with the bitstream of an FPGA attackers can change the circuit that the FPGA implements. Trojans can be added to a bitstream at any point in the build process: they could be added by a malicious actor with access to the source code, they could be added to the bitstream by a compromised compiler, or they could be added directly to an unencrypted bitstream by tampering with the file.

1.1.3 Reverse Engineering FPGA bitstreams

Before discussing the reasons to reverse engineer FPGA bitstreams we first need to define what it means to reverse engineer an FPGAs bitstream. Since the bitstream defines the circuit that an FPGA implements, it stands to reason that if you have a bitstream and you know its formatting then you can analyze it to determine the circuit it programs an FPGA to. This is what we mean by reverse engineering a specific bitstream. However, this assumes that the bitstream format is known, in other words you know which bits in the bitstream configure which components on the FPGA. If you do not know the bitstream format, then it is first necessary to reverse engineer the bitstream format. In other words, to figure out how bits in the bitstream map onto components of the FPGA. It is necessary to understand the distinction between reverse engineering a specific bitstream, and reverse engineering the bitstream format of an FPGA. The former cannot be done without first doing the latter. This Thesis focuses on methods for reverse engineering the bitstream format.

1.1.4 Why reverse engineer FPGA bitstreams

Reverse engineering FPGA bitstreams gives designers another way of checking the bitstream for Trojans. Trojan circuits can be very difficult to find through traditional validation testing, they are often dormant by default and only triggered under very specific circumstances chosen by the attacker. Reverse engineering an FPGAs bitstream is an alternative way of detecting Trojans, it allows designers to check that the circuit contains only the intended logic and nothing more. However, bitstream Trojan detection is made more difficult because FPGA manufacturers do not publish the formatting of the bitstreams. In order to find Trojan circuits in bitstreams the format of the bitstream must first be reverse engineered.

1.1.5 How have FPGAs been REd in the past

There are several public examples of successful FPGA bitstream reverse engineering efforts [2, 3, 7, 8, 9], a few relevant ones will be detailed in Chapter 2. However, each methodology is tied either to a specific FPGA Architecture or a specific manufacturers toolchain. FPGA bitstream reverse engineering requires a high level of control over the configuration of FPGA components. Typically, this high level of control is gained by using specialty features of toolchains to modify designs after the placement step. This means that the tools developed for reverse engineering the FPGA bitstream end up being tied to whichever toolchain they are targeted to and would require significant effort to port to a different toolchain.

1.1.6 What does it mean to RE FPGAs abstractly

Our goal was to come up with FPGA bitstream reverse engineering methodologies that are not tied to a specific FPGA or toolchain. The intent is to create abstract methodologies that are implemented in VHDL and do not rely on any macros or functions specific to any one toolchain. This allows the method to be easily ported to different FPGAs and toolchains. The methodology defined in this Thesis defines an abstract way to reverse engineer the LUT programming bits of an FPGA. It is not reliant on any specialty functions of any toolchain and therefore can easily be implemented across multiple FPGAs and toolchains. Furthermore no specialized knowledge of the FPGA architecture is needed, the only necessary information

is the size of LUTs and number of LUTs on the target FPGA.

1.1.7 What are the benefits of an abstract approach

Implementing abstract methods that are not tied to specific toolchains would allow new FPGA's to be reverse engineered with significantly less effort. A comprehensive abstract bitstream reverse engineering methodology could serve as the foundation for higher-level Trojan detection tools.

1.2 Outline

In Chapter 2 we outline the necessary background knowledge on the general structure of FPGAs and the typical design flow for generating bitstreams. We will also cover some of the existing FPGA bitstream format reverse engineering methodologies. Chapter 3 discusses how our methodology works as well as some of the challenges that the abstract approach comes with. Chapter 4 details the testing method and results. Finally, Chapter 5 gives a conclusion and outlines potential areas of improvement and expansion for the future.

Chapter 2

Background

In this chapter we first discuss the general architecture of FPGAs as well as the typical design flow for bitstream generation. Then we go over prior work on the topic of FPGA bitstream reverse engineering.

2.1 FPGA Background

2.1.1 FGPA Architecture

Figure 2.1 shows the typical Island-Style FPGA Architecture. It consists of logic blocks (LB) connected with reprogrammable switch-boxes (SB) which allow any point on the FPGA to be wired to any other point in the FPGA. Depending on the specific FPGA architecture the logic blocks contain several reprogrammable Look-Up-Tables or LUTs allowing them to implement any desired logic circuit. Many FPGAs also have blocks that contains specialized components like Digital Signal Processing (DSP) modules for example. The methodology presented in this paper focuses on standard logic blocks and reverse engineering the locations of the LUT programming bits in the bitstream.

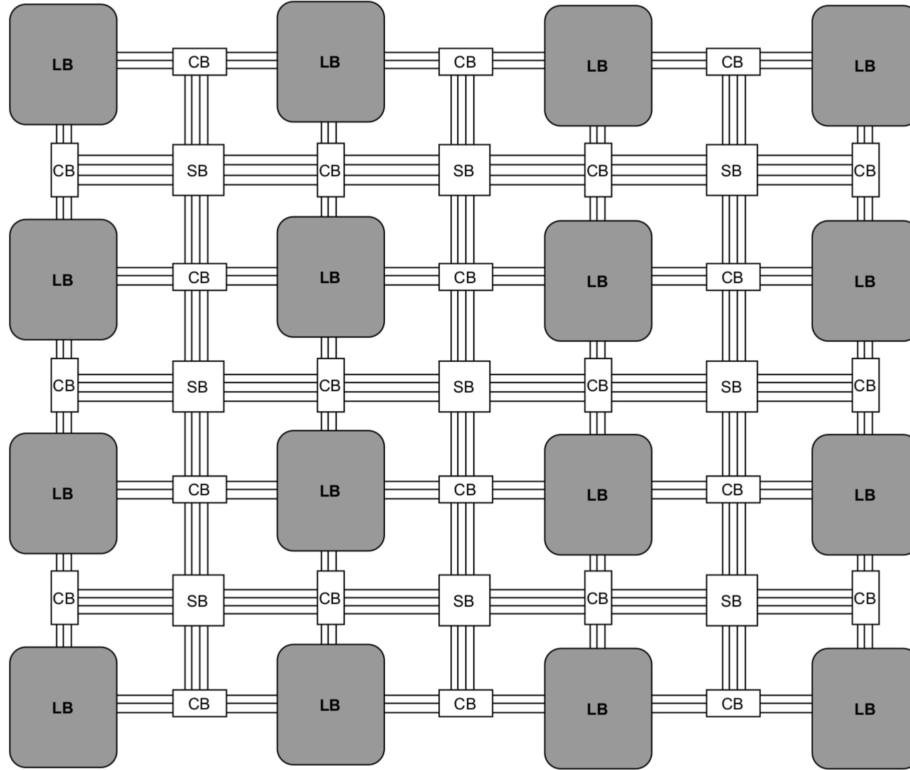


Figure 2.1: FPGA Island Style Architecture

2.1.2 Logic Blocks

Figure 2.2 shows a logic block from the Xilinx 7 Series FPGA architecture. We can see in the diagram that it is composed of four 6 input LUTs, built in carry logic for efficiently implementing arithmetic logic, and eight storage elements that can hold the outputs of the implemented logic. The structure of logic blocks varies depending on the FPGA Architecture, but they all follow the same general structure of having some combination of LUTs, storage elements, and some form of arithmetic carry chain. Some architectures like the Cyclone V, instead of only having a carry chain have dedicated full-adders [6]. The LUTs are the elements critical to allowing an FPGA to arbitrarily implement any logic circuit.

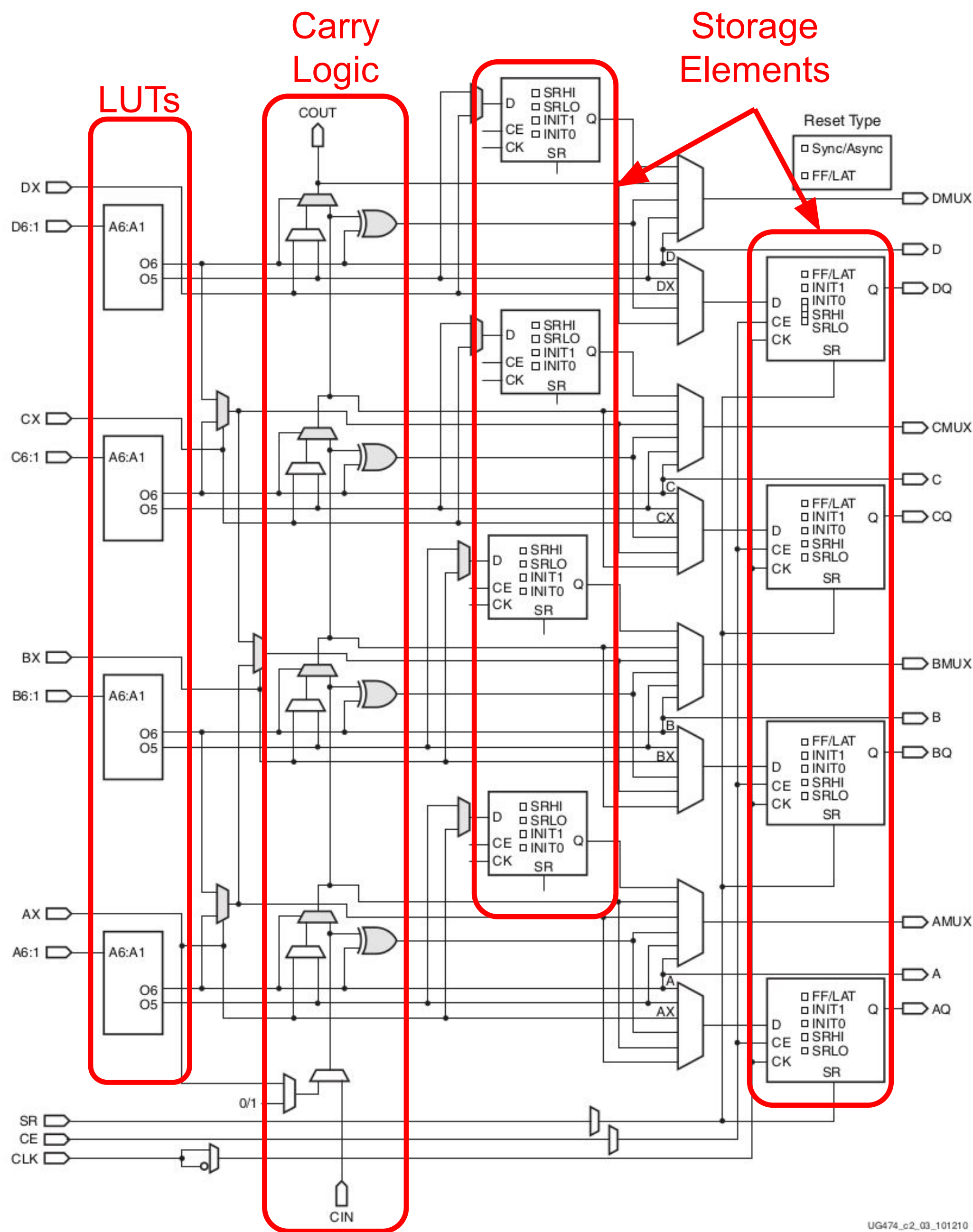
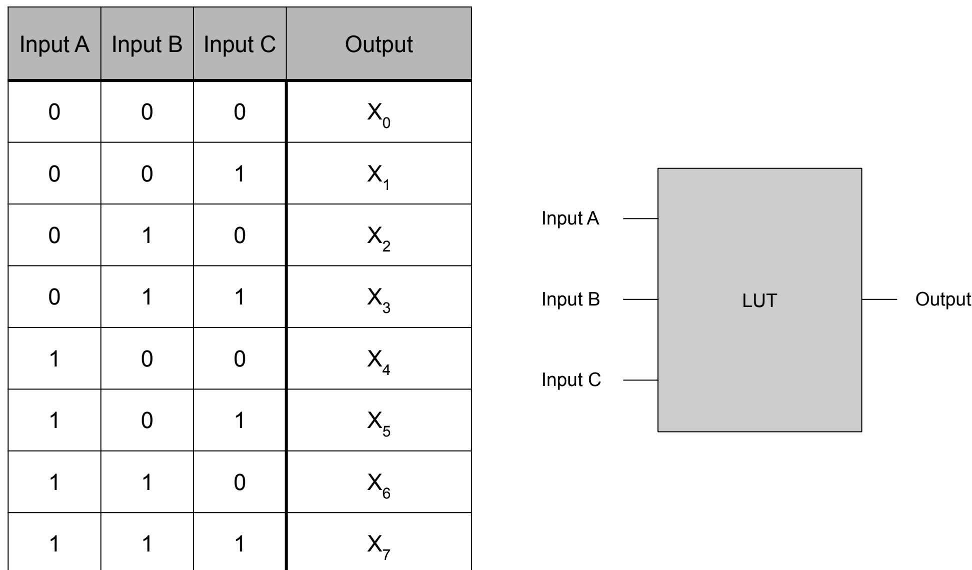


Figure 2.2: Example Logic Block from 7 Series FPGAs CLB User Guide UG474 [10]

2.1.3 Look-Up-Tables

LUTs are the core component of FPGA logic, they are used to create arbitrary combinational logic. A LUT with N inputs can be programmed to implement any N input logic function. Figure 2.3 shows a 3 input LUT. The Outputs X_{0-7} are reprogrammable memory cells which

can be set to a value either 0 or 1. We can see that if the value of the inputs $\{A, B, C\}$ are $\{0, 0, 1\}$ then the LUT will output the value that memory cell X_1 is set to. The values of these memory cells are configured by the bitstream. The goal of the methodology presented in this Thesis is to determine which bits in the bitstream program the LUT memory cells.



Input A	Input B	Input C	Output
0	0	0	X_0
0	0	1	X_1
0	1	0	X_2
0	1	1	X_3
1	0	0	X_4
1	0	1	X_5
1	1	0	X_6
1	1	1	X_7

Figure 2.3: Three Input LUT

2.1.4 LUT Decomposition

Modern LUTs are designed to be decomposable, meaning that a larger LUT can be broken down into multiple smaller LUTs. LUTs are designed this way to increase total logic utilization, making the FPGA more cost effective. For example, notice the LUTs in figure 2.2 have two outputs. This is because the 6 input LUTs of the 7 Series architecture can be broken into two seperate 5 input LUTs, which can each be programmed to a unique function so long as the functions share inputs. LUT decomposition is relevant to the methodology presented in this paper due to the complications it causes for reverse engineering. When reverse engineering LUT programming bits the ideal is to be able to arbitrarily configure the LUT bits because the more control you have over how the LUTs are configured the easier it is to find a correlation between the configuration and the bitstream. However, the compiler will often attempt to optimize the functions programmed to the LUTs to reduce the size of

the circuit. It is a standard feature for FPGA compilers to allow the user to disable logic optimization. We found that even with logic optimization disabled compilers will often still decompose LUTs whose functions do not utilize all of the LUTs inputs. This means that when we are attempting to reverse engineer the LUT programming bits we are limited to programming them with equations that cannot be reduced to using fewer inputs.

2.1.5 FPGA design flow

Figure 2.4 shows the general sequence for FPGA bitstream generation design flow. The process typically begins with a Hardware Description Language (HDL) input defining the circuit to be implemented. The HDL is typically written in either VHDL or Verilog. This HDL is then synthesized into a RTL (Register Transfer Level) netlist representation. The components of this netlist are then mapped to physical components on the target FPGA in the placement step. Then the placed components are connected in the routing step. The final step of the process is to take the placed and routed design and generate a bitstream with which to program the FPGA.

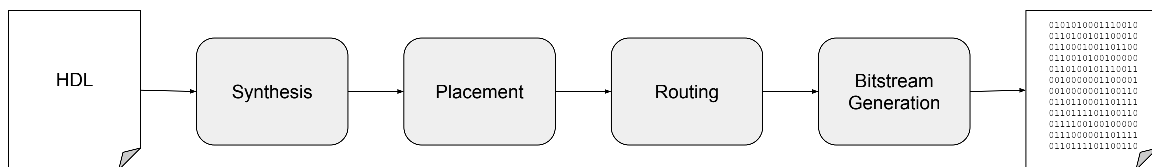


Figure 2.4: FPGA Design Flow

2.2 Prior Work

2.2.1 Project X-Ray

Project X-Ray [8] is a reverse engineering effort targeting Xilinx 7-Series FPGAs. It is a part of the F4PGA Project (formerly SymbiFlow) which intends to create a fully opensource FPGA compiler. Project X-Ray is built around Vivado, Xilinx's compiler. Their approach consists of a collection of Fuzzers, which are scripts designed to target a specific component on the target FPGA and generate bitstreams to find the bitstream bits that control the

targeted component. One of the Fuzzers in Project X-Ray targets the LUT programming bits. This LUT Fuzzer functions by creating a matrix of LUTs using Verilog where the LUTs are instantiated by using the Vivado LUT macros. The Fuzzer works by first synthesizing and placing the LUT matrix, then it uses specialized Vivado commands to manually change the LUT configuration bits on the placed design and regenerate new bitstreams. This allows the Fuzzer to avoid additional noise from routing changes by ensuring that the only bits changed between bitstreams are LUT bits. Furthermore, it uses the Vivado LOCK_PINS property to force the compiler to not reorder the LUT bits, thereby avoiding the complications of input pin reordering which are detailed in Chapter 3.

Project X-Ray is the most thorough implementation of FPGA bitstream reverse engineering that is available publicly. It has successfully mapped the majority of the 7-Series Xilinx FPGA bitstream architecture. However, the codebase is specifically tied to the Xilinx Vivado toolchain. The methodology presented in this paper only targets the LUT programming bits, but it does it in a way that does not require the use of Vivado specific macros or commands, thereby allowing the method to be easily used across many different FPGAs and toolchains.

2.2.2 Logarithmic-Time FPGA Bitstream Analysis

The paper "Logarithmic-Time FPGA Bitstream Analysis: A step towards JIT Hardware Compilation" [2] describes an abstract reverse engineering method at a high level. The paper defines an algorithm that considers each component in the FPGA a "Programmable Point" with an associated bit or set of bits in the bitstream. This algorithm allows for the matching of each Programmable Point to its associated bitstream bit or bits in logarithmic time complexity, meaning that to solve N programmable points it only needs to generate $log_2(N)$ bitstreams. However, this method requires precise control over the Programmable Points which can be difficult or impossible to attain with some FPGA compilers. The paper is able to demonstrate the effectiveness of the algorithm on Xilinx FPGAs by using XDL. XDL is a Xilinx specific language that allows for low-level control of the bitstream, however Intel does not have an equivalent tool. Therefore at the time of writing it is not possible to use the defined algorithm on Intel FPGAs. However, the fundamental principle proposed

in the paper is to set the FPGA bitstreams in a way that results in the target bitstream bits exhibiting unique patterns over the course of several bitstream generations. The exact nature of the algorithm allows for logarithmic time performance solving the bitstream bits. The methodology proposed in this paper uses the same principle for matching bitstream bits to the FPGA components, however rather than using an exact configuration to achieve log time performance we use a randomized approach to configure bits. This makes the algorithm possible to implement on FPGAs where exact control is not possible, but results in a slight loss of performance, only achieving log time performance in the average case.

2.3 Summary

We discussed the basic architecture of FPGAs specifically focusing on LUTs which are the primary target of the reverse engineering methodology presented in this Thesis. We also discuss two public examples of reverse engineering methodologies and outline how they differ from our method.

Chapter 3

Methodology

3.1 Generic LUT Mapping

Reverse Engineering of the LUT bits requires control over the manipulation of LUT equations. In [8, Project X-Ray] LUT instantiation is done using HDL macros provided by the tool chain. This means that the HDL written for one tool chain would not necessarily work on another tool chain. Instead, we use VHDL case statements to infer LUTs. This means that the code is not tied to any specific toolchain. It does however come with some limitations which will be discussed in the Challenges section.
The general idea of the methodology is to use generic VHDL case statements to infer the instantiation of LUTs. Then we wire the case statements together in a way exploits the timing optimization of FPGA compilers to attempt to get our design consistently placed in the same location on the FPGA.

3.1.1 Challenges

Using generic VHDL to instantiate LUTs comes with several challenging limitations. The methodology we developed is specifically designed to overcome these limitations, so its necessary to understand these limitations in order to understand why our methodology is designed the way that it is. Therefore, we will begin by outlining the limitations of using generic VHDL to instantiate LUTs.

Input Pin Reordering Input pin reordering refers to an optimization step done during the place-and-route stage of compilation. The compiler will often change the order of LUT input pins to reduce timing delays. Input pin reordering makes the reverse engineering process significantly more complicated, because it means that the values written to VHDL case statements will not be implemented the same way on hardware, preventing direct control of LUT bit assignment. Figure 3.1 shows an example of how this could work, demonstrated with a 2 input LUT. In the example we use a VHDL case statement to program a 2 input LUT with the values X_{0-3} but the compiler then swaps the inputs of the LUT. This results in the values X_1 and X_2 swapping positions in the physical LUT memory cells. The behavior of the circuit is not affected as the LUT will still output X_1 when A=0 and B=1, but the physical LUT bits have not been programmed the way that we intended.

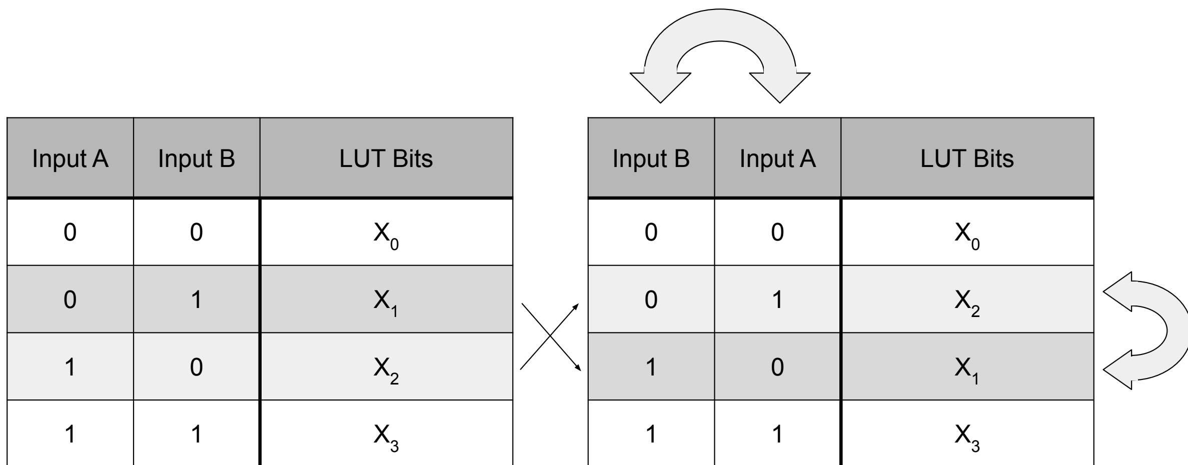


Figure 3.1: Input Pin Reordering

Logic Optimization Compilers will attempt to reduce the number of logic elements utilized when possible. This poses a challenge to reverse engineering because it means that it is not guaranteed that structures implemented in HDL will be implemented the same way on the physical FPGA. It is a standard feature for FPGA compilers to allow logic optimization to be disabled, however even with logic optimization disabled case statements whose equations do not utilize all inputs will often be mapped to decomposed LUTs. Therefore, it is necessary to only program LUTs with equations that are irreducible.

LUT Packing Some FPGAs featuring complex LUT architectures are capable of packing two LUTs that share multiple inputs and have the same equation into a single LUT. Most LUTs architectures are designed to be decomposable, meaning that a single larger LUT can also function as two or more smaller LUTs. To achieve this LUT architectures generally support having two outputs. Some LUT architectures can exploit this to allow them to pack together LUTs that have the same equation and share most of their inputs. This behavior was observed with the Cyclone V LUT architecture.

3.1.2 Method

Hamming Weight The problem of Input Pin Reordering can be solved by only programming the LUTs with equations that do not change if the input pins change. Equations meet this criterion if all programmable bits with the same hamming weight are set to the same value. The hamming weight of a LUT bit is the number of ones in that bit's address. Table 3.1 shows the hamming weight of the bits of a two input LUT. The intuition follows that if you reordered the inputs i.e. swapped columns A and B in the table, the hamming weight would not be affected. Therefore, if all the bits with an address hamming weight of 1 (bits B_1 and B_2) are programmed to the same value, then swapping the inputs would not change the values programmed to the physical LUT even though the values programmed to B_1 and B_2 have technically changed positions.

A	B	Out	Hamming
0	0	B_0	0
0	1	B_1	1
1	0	B_2	1
1	1	B_3	2

Table 3.1: LUT Bit Hamming Weight

XOR/XNOR Functions The XOR and XNOR functions are the ideal functions to work around both the input pin reordering problem and the logic optimization problem. Both XOR and XNOR functions are irreducible and both functions meet the hamming weight criterion, meaning that the LUT programming bits do not change if the inputs of the equation

are reordered. The LUT programming bits for an XOR equation are also the exact inverse of an XNOR equation with the same number of inputs, this makes them especially useful for mask generation, as described later.

3.1.3 LUT Interconnect Patterns

We use generic VHDL case statements to instantiate individual LUTs, however this still leaves the question of how to connect case statements together to instantiate multiple LUTs. The goal is to maximize LUT utilization per bitstream generation, because any LUTs left unprogrammed would not be solved. We also have the secondary goal of trying to get the compiler to place consecutive designs in the same location on the FPGA, thereby reducing noise from routing changes. The interconnect patterns we tested try to achieve this goal by exploiting the compilers timing optimization during place-and-route. The idea is that if we chain together LUTs creating a massive combinational logic structure the compiler will find an optimal placement with minimum timing delays. The assumption being that the compiler would find the same optimal placement from run to run. This will be discussed further in the Results section, but we found that in the majority of tests this assumption proved to be correct though there were some cases where routing did change. Over the course of the research we tested three different approaches to connect the LUTs, the Row/Column approach, the Snake approach, and the Modified Snake approach.

Row/Column The first method tested simply instantiates a matrix of LUTs in which the first column's LUT inputs are initialized by I/O and the output of each LUT is wired to the inputs of LUTs near it in the next column. This method was simple to implement, however it had several drawbacks. It performed well at getting the compiler to place designs consistently, which helps reduce noise. However, it suffered from the problem of LUT packing with some FPGAs. The main issue with this methodology was that it required some tuning to the specific architecture of the FPGA. Most FPGAs group LUTs together into blocks that share routing resources. These are commonly referred to as Slices. For some FPGAs the Row/Column method required that the size of columns was a multiple of the number of LUTs in a slice in order to maximize LUT utilization. For example, the Xilinx 7 Series

FPGA architecture has 4 LUTs per Slice, and we found that the size of a Column had to be a multiple of 4 to get maximum LUT utilization. This meant that the number of LUTs instantiate could only be changed 4 LUTs at a time. This is a problem because often compilers are not capable of getting 100% LUT utilization, but they could place designs that used nearly 100% of LUTs minus 1-5 or so. This means that the Row/Column method was not ideal for getting the absolute maximum amount of LUT utilization possible because if you have an FPGA that is capable of filling all LUTs except 1, then with the Row/Column method you need to remove the entire last column. The method also requires the user to know how many LUTs are in a slice for the target FPGA. This is typically easy information to find, but our goal is to make FPGA REing as simple as possible, so ideally we reduce the need for specialized knowledge wherever possible.

Algorithm 1 2-D Row/Column Pattern

1: **function** $LUT(A1, A2, ..., AN)$
2: A = binary2integer(AN, . . . , A2, A1)
3: case A = 0: Out ⇐ MC(0) value
4: case A = 1: Out ⇐ MC(1) value
5: case A = 2: Out ⇐ MC(2) value
6: ⋮
7: case A = 2^N-1: Out ⇐ MC(2^N-1) value
8: **end function**
9: **function** 2D_Row_Column_Map(R, C, N)
10: ▷ R: Rows, C: Columns, N: Number of LUT inputs
11: ▷ LUTs near perimeter handled as special cases
12: **for** i = 2 to C **do**
13: **for** j = 1 to R **do**
14: **if** N is even **then**
15: $LUT_{j,i}(LUT_{j-N/2+0,i-1}(),LUT_{j-N/2+1,i-1}(),\cdots,\cdots,LUT_{j+N/2+(N-1),i-1}())$
16: **else if** N is odd **then**
17: $LUT_{j,i}(LUT_{j-N/2+1,i-1}(),LUT_{j-N/2+2,i-1}(),\cdots,\cdots,LUT_{j+N/2+N,i-1}())$
18: **end if**
19: **end for**
20: **end for**
21: **end function**

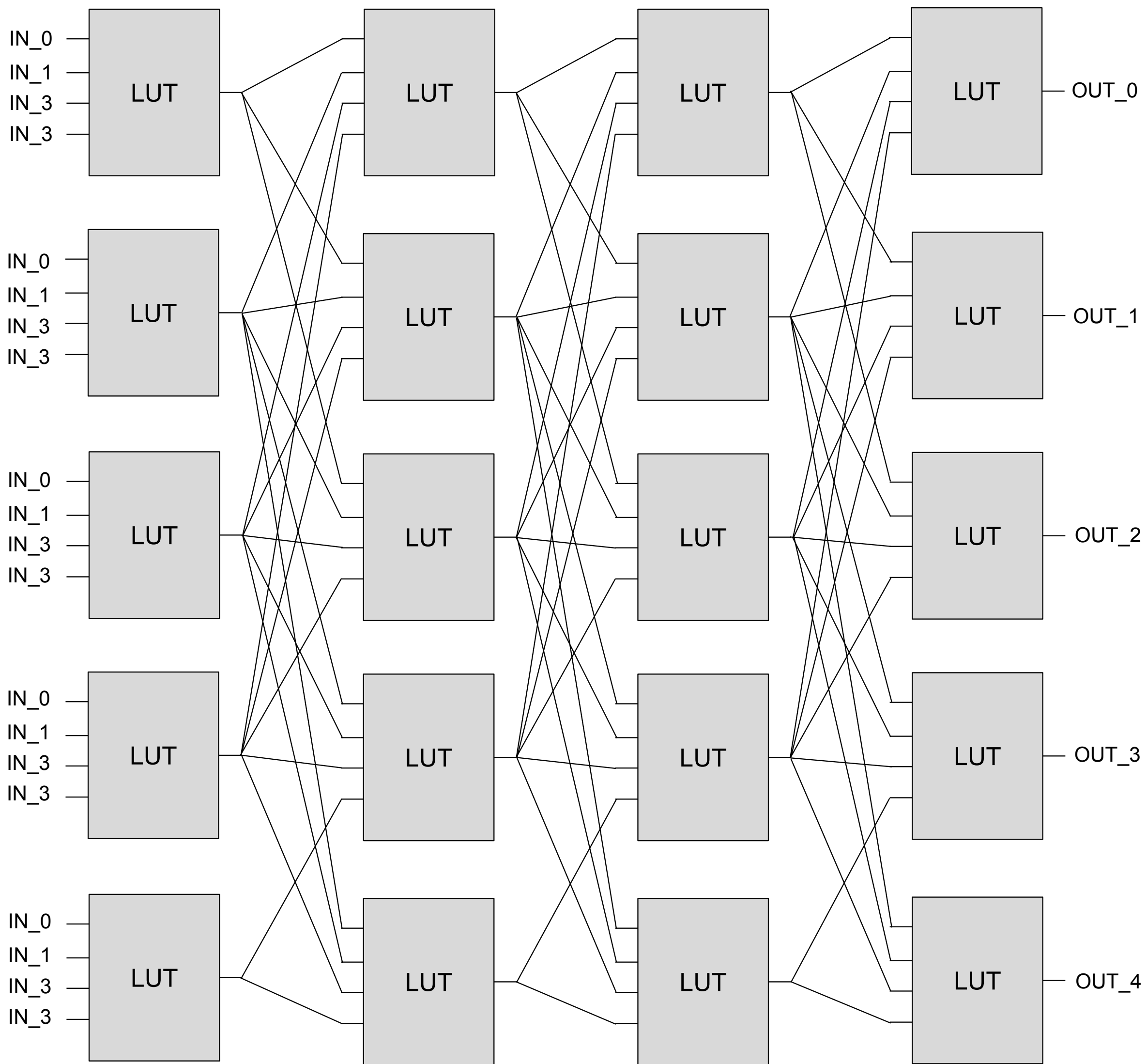


Figure 3.2: Row/Column Method

Linear Snake The second methodology tested was the Linear Snake method. So-called because it wires LUTs together in a chain to form a "Linear Snake" of LUTs. The inputs to an N input LUT are wired to the outputs of the prior N LUTs in the chain, as shown in figure 3.3. The LUTs at the start of the chain are initialized by external I/O. We found that this methodology preformed as well as the Row/Column method at getting consistent placement between bitstream generations. It also has the benefit of more fine-grained control of the number of LUTs over the Row/Column method and does not require knowledge of the

number of LUTs in a slice. However, it is still susceptible to the problem of LUT packing with some FPGAs.

Algorithm 2 Linear Snake Pattern

```
1: function LINEAR_SNAKE_MAP(N, N)
2:                          ▷ X: Number of LUTs, N:Number of LUT inputs
3:                          ▷ Initial LUT_j (j < N) handled as special cases
4:     for i = N+1 to X do
5:         LUT_j(LUT_{j-1}(),LUT_{j-2}(),···,LUT_{j-N}())
6:     end for
7: end function
```

Modified Linear Snake The Modified Linear Snake pattern was created to address the issue of LUT packing. As stated in Section 3.1.1 Challenges, LUT packing occurs with some LUT architectures if LUTs programmed with the same equation share too many inputs. The mask generation step of the methodology requires that all LUTs be programmed to the same values, so to prevent LUT packing we must minimize the number of inputs shared between LUTs. This is what the Modified Snake method seeks to accomplish. Rather than simply using the prior N LUTs to drive the inputs of any given LUT in the chain, the Modified Snake method uses the series of Triangular Numbers to determine which LUTs to connect.

Algorithm 3 Modified Snake Pattern

```
1: function MODIFIED_SNAKE_MAP(X, N)
2:                          ▷ This is a modification to the Snake Map algorithm
3:         ▷ LUT I/O ordering changed to leverage partially decomposable LB architecture
4:             ▷ Assuming N=6 (LUT6); Initial LUT_j (j < 3N-1) handled as special cases
5:     for j = 3N-1 to X do
6:         LUT_j(LUT_{j-1}(),LUT_{j-2}(),LUT_{j-4}(),LUT_{j-7}(),···,LUT_{j-(N(N-1))/2)-1}())
7:     end for
8: end function ..........................................................
```

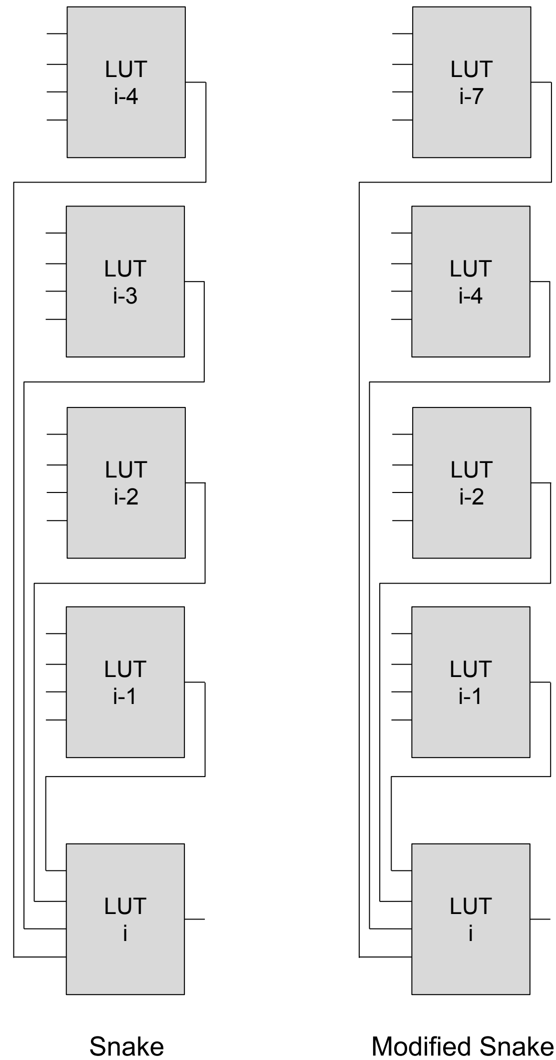


Figure 3.3: Snake Methods

3.1.4 Solving the LUTs

LUT Mask Generation The first step of the process is to identify which bits in the bitstream program the LUTs. The process for obtaining the LUT mask is very simple. First two bitstreams are generated, one in which all LUTs are programmed to the XOR equation and another where all LUTs are programmed to the XNOR equation. Then the xor of the two bitstreams is taken, the result is the LUT mask. The logic follows that the xor of the two bitstreams will have 1's for any bits that changed from the first bitstream to the second, since

the LUT equations that were programmed to the FPGA are inverses of each other. If all LUTs were utilized then all the LUT bits will appear as 1's in the mask. However, they may also contain "noise" in the form of bits that changed value but do not belong to the LUTs. Common sources of this kind of noise include routing bits or differences in the bitstreams header metadata. This noise can be significantly reduced if the chosen LUT interconnect pattern succeeds in getting the tool to place the design in the exact same location, and the next step of the process is also effective at eliminating random noise.

Figure 3.4 shows an example mask generation. For the example we imagine an extremely small FPGA which has only two LUTs, LUT A and LUT B. Each LUT only has two inputs, meaning each LUT has 4 programmable bits. The bitstream is also extremely short having only 11 bits in total, 8 of which belong to the LUTs. Figure 3.4 shows how in the first bitstream LUT A (bits B_{1-3}) and LUT B (bits B_{4-7}) are programmed to XOR functions, and in the second bitstream they are programmed to XNOR functions. We then compute the xor of both the bitstreams to generate the mask. Notice that in this example there are two unwanted noise bits that made it into the mask, bits B_9 and B_{10}. The next section will show how these noise bits are eliminated.

LUT A		
Input A	Input B	Programmable Bits
0	0	B_0
0	1	B_1
1	0	B_2
1	1	B_3

LUT B		
Input A	Input B	Programmable Bits
0	0	B_4
0	1	B_5
1	0	B_6
1	1	B_7

	Bitstream Bits										
Bitstreams	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
XOR Bitstream	0	1	1	0	0	1	1	0	0	1	0
XNOR Bitstream	1	0	0	1	1	0	0	1	0	0	1
XOR ⊕ XNOR = Mask	1	1	1	1	1	1	1	1	0	1	1

Figure 3.4: Mask Generation

Pseudo-Binary Search The Pseudo-Binary Search algorithm works off the principle that if we repeatedly generate bitstreams in which each LUT is randomly configured to either an XOR or XNOR function, then given enough bitstreams each individual LUT will have been programmed to a unique sequence of XOR/XNOR equations. Once enough bitstreams have been generated that all LUTs have a unique sequence we can then group bits in the mask that exhibited the same pattern. We expect bits that belong to the same LUT to exhibit the same pattern, with the exception that the XOR vs XNOR bits will have inverse patterns relative to each other. Meaning that half of a LUTs bits will exhibit the inverse pattern of the other half. For example, we would expect that for a 6 input LUT (with 64 bits) there would be two groups of 32 bits that all exhibited the same patterns and the patterns of the two groups are the inverses of each other.

Continuing the prior example from Figure 3.4, Figure 3.5 shows how bits belonging to the same LUT are paired with each other and how noise is eliminated. The table at the top of Figure 3.5 shows four bitstreams generated. In each bitstream LUT A and LUT B were randomly set to either the XOR or XNOR function. Below the table we see a diagram

showing the progression of the algorithm as it processes each additional bitstream. It starts at the top with all the mask bits, in this example it is all bits except B_8. However, in a real world bitstream the mask eliminates a significant portion of the bitstream. Then we begin processing the first random bitstream finding what each bit of intrest was programmed to. Using this information the original set is split in two, the first set contains all the bits programmed to 0 in the bitstream, the second all the bits programmed to 1. This is repeated with the next bitstream, now splitting set 0 into set 00 and 01 where each bit was set to 0 in the first bitstream and then to 0 or 1 respectively in the second bitstream. In this stage we can eliminate the first noise bit B_9, because it is in a set with fewer than 2 members. We know that the FPGA has 2 input LUTs, and we expect half of the bits of each LUT to exhibit the exact same pattern. In other words, for LUTs with M bits we expect sets with no fewer than M/2 bits that have the exact same pattern. This means that any set which contains fewer than M/2 bits can be eliminated, meaning for our example sets with fewer than 2 bits.

	Bitstream Bits										
Bitstreams	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
RND 1	0	1	1	0	1	0	0	1	1	0	0
RND 2	1	0	0	1	0	1	1	0	1	0	1
RND 3	0	1	1	0	0	1	1	0	0	1	0
RND 4	0	1	1	0	1	0	0	1	1	1	1

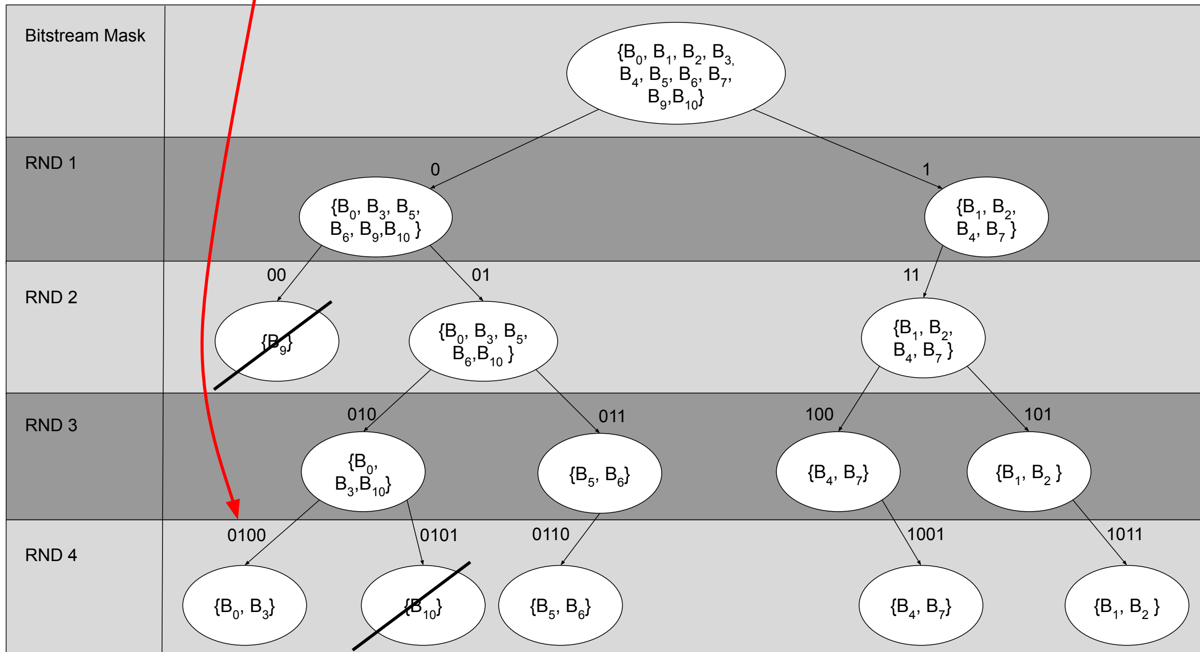


Figure 3.5: Pseudo-Binary Search

Algorithm 4 Pseudo-binary Search

```
1: procedure Sort(X,N,Mask)
2:     var R, Tmp : table
3:     var B : bitarray
4:     var T, F, TmpT, TmpF : set
5:     var newkey : string
6:     var complete : boolean ← False
7:     while not complete do
8:         Tmp ← new table                                   ▷ initialize to empty table
9:         B ← GenerateRandomBitstream(X, N)
10:        T ← GetSetBitIndexes(Mask and B)    ▷ returns a Set of the indexes of every
           bit set to 1
11:        F ← GetSetBitIndexes(Mask and not B)
12:        if Rif empty then
13:            Tmp["T"] ← T
14:            Tmp["F"] ← F
15:        else
16:            complete ← True
17:            for each (key, value) in R do
18:                TmpT ← value ∩ T
19:                if GetLength(TmpT) >= N^2/2 then
20:                    newkey ← key + "T"
21:                    Tmp[newkey] ← TmpT
22:                end if
23:                TmpF ← value ∪ F
24:                if GetLength(TmpF) >= N^2/2 then
25:                    newkey ← key + "F"
26:                    Tmp[newkey] ← TmpF
27:                end if
28:                if GetLength(TmpT) > N^2/2 or GetLength(TmpF) > N^2/2 then
29:                    complete ← False
30:                end if
31:            end for
32:        end if
33:        R ← Tmp
34:        Delete(Tmp)
35:    end while
36:    return R
37: end procedure
```

Bit matching Upon completion of the Pseudo-Binary search, we will have sets of bitstream bits in which all bits belong to the same LUT but each set only contains half the bits of

a LUT. This is because we used the XOR and XNOR functions to program the LUTs so for any one configuration, half of a LUTs bits were set to 1's and the other half set to 0's. Because the XOR and XNOR functions are inverses of each other we expect that the two halves of the LUT exhibit inverse patterns of each other. So, we can form completed LUTs by matching the sets whose patterns are inverses of each other. For example, in Figure 3.5 the set B_0,B_3 had the pattern 0100 whose inverse is 1011 which matches set B_1,B_2. So we then combine these sets to form a completed LUT B_0,B_1,B_2,B_3.

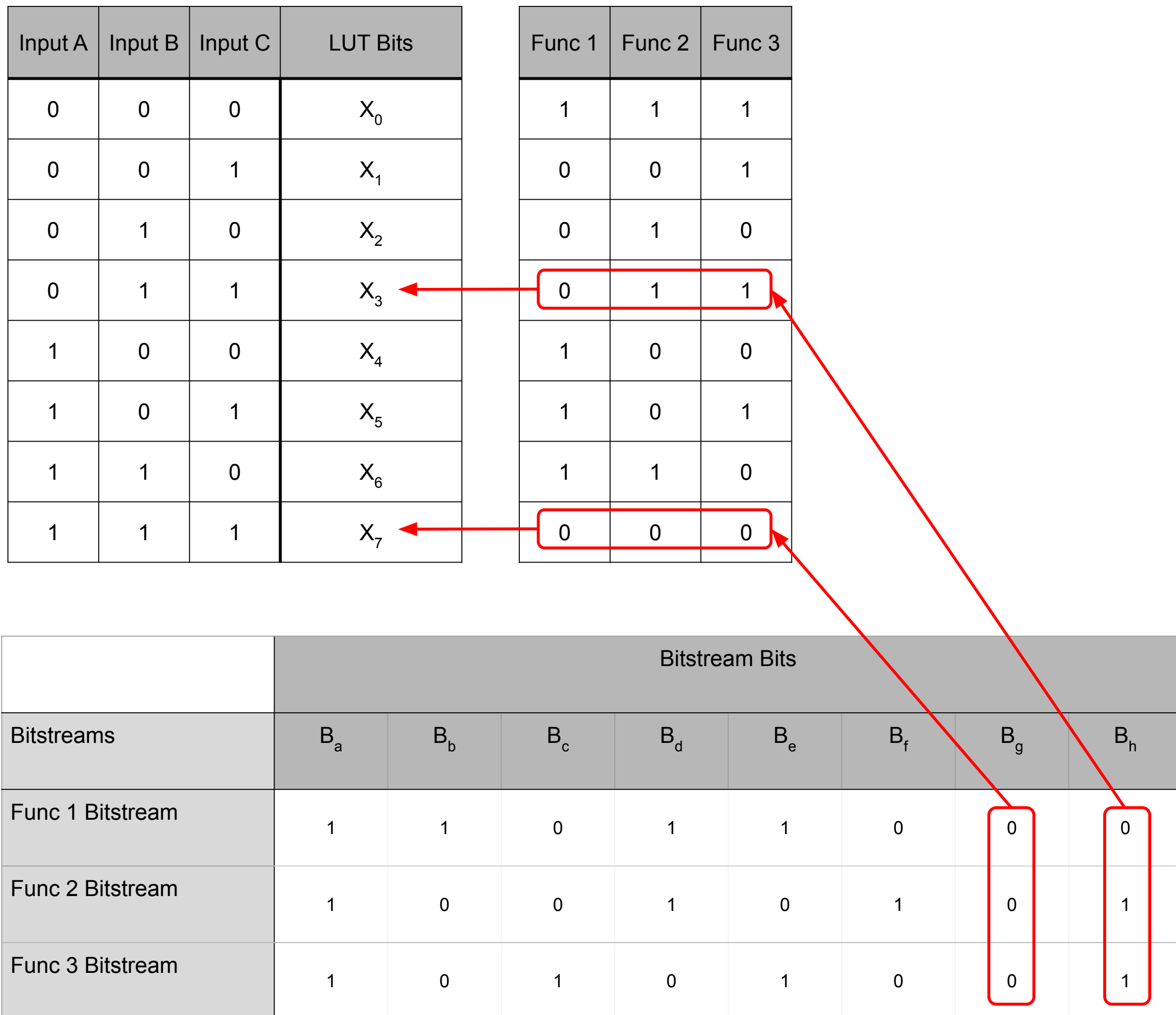
While using the patterns inverse to match sets is simple and generally reliable, it does have one problem. Using the inverse fails if the LUT was not utilized in one or more of the bitstreams used to create the pattern. This is because if the LUT is not utilized then the LUT is generally set to either all zeros or all ones by default. Meaning that both halves of the LUT are set to the same value, so the inverses of the pattern will not match. This is only a problem if it is not possible to obtain 100% utilization on the FPGA, and the LUT interconnect pattern does not succeed at getting the compiler to consistently place the design.

One alternative method for matching the sets is to use the concept of hamming distance, which looks at the number of bits that differ between the patterns. If the exact inverse does not result in a match, then we can pair the sets with whichever set whose pattern has the greatest hamming distance. Effectively pairing sets whose patterns have the most number of bits that are inverses out of all the sets.

3.1.5 Sorting LUT bits

The process detailed above solves for which bits belong to the same LUT, however this is not enough information to determine what logic equation any given LUT is programmed to. To fully extract LUT logic equations we must determine the order in which each LUT bit fits into the truth table. We refer to this process as sorting the LUT bits. For an N input LUT this is done by simply programming the LUTs with the column values of an N input truth table until we observe N unique patterns in the bits belonging to the LUT. Once these N unique patterns are observed we arbitrarily order them 1-N and use the numerical value that each LUT bit exhibited to place it at that index in the truth table.

This is much easier to understand with an example. In 3.6 we have a 3 input LUT whose bits we have solved $\{B_a, B_b, B_c, B_d, B_e, B_f, B_g, B_h\}$ but we do not know the order of these bits in the LUT. To sort them we generate three bitstreams programming the LUT with the values Func 1-3 as shown in Figure 3.6. Notice that the programmed values are not exactly the same as the columns of the truth table. The most significant and least significant bits are swapped. We must do this because otherwise the function would be reducible and likely the compiler would decompose the LUT to a smaller one or simply remove it entirely. Once the bitstreams are generated we look at the values that the bits B_{a-h} exhibited. We expect that the LUTs bits will have been programmed to three distinct patterns, one for each input. If this is the case, we can use the pattern each bit exhibited to place them in the corresponding location in the truth table, keeping in mind that we swapped the most significant and least significant bits. Bit B_h for example had the binary value "011" which is 3 in decimal, so we sort it into index 3 in the truth table. However, some FPGAs use inverted logic to program LUT memory cells which would result in us incorrectly reversing the order of the bits. To check for this we look at the value that the most significant bit (MSB) and least significant bit (LSB) were set to in the XOR mask bitstream. Since we know that the LUT was programmed to the XOR function in this bitstream we expect the values of the MSB and LSB to be 0s. If they are not, then we know the FPGA uses inverted logic, and we reverse the LUTs bit order to compensate. It should be noted that because of input pin reordering there is no guarantee that the input ordering of our sorted LUT matches the physical pin order of the physical LUT on the FPGA. In fact the true pin ordering is much more likely to differ from out virtual ordering. However, the true physical order of the inputs does not really matter, what matters is that we can distinguish between the inputs to reconstruct a logic equation. Furthermore, also because of input pin reordering it may take more than N bitstreams for us to observe N unique patterns in the LUTs bits because the compiler may reorder the inputs so that we observe a duplicate of a previous pattern. If this is the case then we generate more bitstreams until we observe all N unique patterns, and we discard any duplicates.



Input A	Input B	Input C	LUT Bits
0	0	0	X_0
0	0	1	X_1
0	1	0	X_2
0	1	1	X_3
1	0	0	X_4
1	0	1	X_5
1	1	0	X_6
1	1	1	X_7

Func 1	Func 2	Func 3
1	1	1
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
0	0	0

	Bitstream Bits							
Bitstreams	B_a	B_b	B_c	B_d	B_e	B_f	B_g	B_h
Func 1 Bitstream	1	1	0	1	1	0	0	0
Func 2 Bitstream	1	0	0	1	0	1	0	1
Func 3 Bitstream	1	0	1	0	1	0	0	1

Figure 3.6: LUT Bit Sorting

Chapter 4

Results

4.1 Test Methodology

To test our methodology we chose a variety of FPGAs from both Xilinx and Intel. We looked up the size and number of LUTs available on each FPGA from their datasheets, as well as the number of LUTs in each logic block for the Row/Column method. Then we attempted to generate bitstreams for each FPGA using the three LUT interconnect methods, instantiating the maximimum number of LUTs on the FPGA. If bitstream generation failed we reduced the LUT count by one and tried again, or in the case of Row/Column reduced the number of rows by one. This was repeated until we had successfully generated all necessary bitstreams for each FPGA for all three LUT interconnect methods. For the FPGAs that did not successfully have full LUT utilization, the entire process was repeated a second time using the next highest number of LUTs that could successfully generate a bitstream.

4.2 LUT Utilization Results

The objective of the LUT interconnect strategies was to make the placement of LUTs consistent accross multiple bitstream generations and to maximize the number of LUTs that could be utilized. The reason that it is important to maximize LUT utilization is that any LUTs that are not programmed will not be solved. To solve for these missed LUTs we repeat the process and slightly change the number of LUTs instantiated to force the compiler to

place the design differently in hopes that the new placement will include the LUTs that were missed. The results of both runs can then be combined to obtain a full mapping. However, because the compiler is not guaranteed to replace the design to include the missed LUTs it is still critical to get as high LUT utilization as possible because this maximizes the chance that all LUTs will be solved in the second run.

Table 4.1 shows the maximum number of LUTs that each method was able to instantiate in a single run. The data shows that the snake based methods either matched or outperformed the utilization of the Row/Column based approach. Furthermore the modified snake method, although it requires additional I/O resources, performed as well as the original snake method while solving the LUT merging issue experienced with the Cyclone V architecture.

FPGA Vendor	Family	Part	LUTs Available	Max LUT Utilization		
				Row/Column Method	Snake Method	Modified Snake Method
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14380	14399	14399
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	18479
Xilinx	Artix	xc7a100tfgg484-2L	63400	63384	63398	63399
Xilinx	Kintex	xc7k70tfbg484-3	41000	40992	40997	40999
Xilinx	Spartan	xc7s15cpga196-2	8000	8000	8000	8000
Xilinx	Spartan	xc7s100fgga484-2	64000	63992	63999	63999
Xilinx	Zynq	xc7z020clg484-1	53200	53192	53199	53199

Table 4.1: Maximum Single Run LUT Utilization Achieved

4.3 Problems

Artificially Restricted FPGAs One problem encountered was that there appeared to be some FPGAs that were being artificially restricted to utilize fewer LUTs than were physically available on the FPGA. This problem was found with several FPGAs from both Xilinx and Altera. Upon further investigation it appears that both manufacturers are officially listing some FPGAs as having fewer LUTs available than there are physically available. Furthermore, all LUTs remain capable of being programmed, there is just an artificial restriction in the compiler to not allow all of the LUTs to be utilized at the same time. For each of these artificially limited FPGAs we were able to find a "larger" FPGA with the exact same packaging and I/O, whose bitstreams looked suspiciously similar, with the exception of the FPGA name in the header data. It appears that FPGA manufacturers are artificially expanding their product line of FPGAs by selling the exact same FPGAs with different names

on the package and using the compilers to restrict the resource utilization allowed on the ”smaller” FPGAs. This causes a problem for our reverse engineering methodology because it requires high LUT utilization, ideally full LUT utilization. It may be possible to RE these FPGAs with our methodology if you manually constrained the input and output pins to guide the placement of the LUTs to different portions of the FPGA, doing repeated runs and combining the result. However, for the purpose of demonstrating this methodology we did not use the FPGAs for which the compiler artificially prevented full LUT utilization.

One of the FPGAs that we suspect to be artificially restricted is the 14400-LUT xc7z007sclg400-1 which is the FPGA found on the Cora-Z7 development board. The data we generated suggested that it actually has 17600 LUTs, and the bitstream looked very similar to that of the Cora-Z10's 17600-LUT xc7z010clg400-1 FPGA. Fortunately, we had a Cora-Z7 on hand that we were willing to risk destroying. So, we tested the theory by generating a simple bitstream for the Cora-Z10 that blinked an LED, tampering with the bitstream header to rename the FPGA part name, and then attempting to program our Cora-Z7 with it. It did not work at first, but after disabling CRC error correction and identifying two other bits that needed to be changed, we were able to successfully program the Cora-Z7 with a bitstream that was generated for the Cora-Z10. However, this is not a thorough enough test to declare that the FPGAs are identical, or to make the same assumption about the other FPGAs that we identified as behaving this way. But it strongly reinforces the hypothesis that the discrepancies we found in our data for these FPGAs stem from vendors artificially restricting LUT utilization in the compiler.

Difficulties Obtaining Full LUT Utilization Even with the more optimized LUT interconnect patterns it was still not possible to obtain full LUT utilization with all FPGAs. With Altera FPGAs it is actually impossible to utilize all of the LUTs in a single bitstream because it always configures a single LUT to all 0's with the output used as ground. This meant that for many of the FPGAs, while it was possible to solve more than 99% of LUTs in a single run it was not possible to get 100%. The solution to this was to simply do a second run using a slighty smaller number of LUTs. By changing the number of LUTs this would change the way that the LUTs were placed on the FPGA resulting in a high probablility

that the LUTs that were not solved in the first run would be solved in the second run. In all cases this was able to yield 100% LUTs solved by doing no more than two runs.

Table 4.2 shows how many total LUTs were solved for each of the FPGAs tested. All the FPGAs except the 8000 LUT Spartan required two runs to solve the full FPGA, as can be seen in the detailed tables for each interconnect method later in this chapter. We combined the results of two runs by taking the union of the sets of LUTs solved in each run. This is a straightforward process, the data for each LUT is an array of bit indices so to combine the result of two runs we simply start with the LUT data from the first run, then parse through the LUTs of the second run, keeping any LUTs with unique values not present in the first dataset and discarding any that are already known.

FPGA Vendor	Family	Part	LUTs Available	LUTs Solved		
				Row/Column Method	Snake Method	Modified Snake Method
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14400	14400	14400
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	18480
Xilinx	Artix	xc7a100tfgg484-2L	63400	63400	63400	63400
Xilinx	Kintex	xc7k70tfbg484-3	41000	41000	41000	41000
Xilinx	Spartan	xc7s15cpga196-2	8000	8000	8000	8000
Xilinx	Spartan	xc7s100fgga484-2	64000	64000	64000	64000
Xilinx	Zynq	xc7z020clg484-1	53200	53200	53200	53200

Table 4.2: LUT Solving Data

4.4 LUT Solving Results

Row/Column Results In order to maximize the number of LUTs utilized with the Row/-Column method it was often necessary to make the column height a multiple of the slice size. For this reason the Linear Snake approaches were able to match or exceed the total LUT utilization of the Row/Column approach on the same FPGA. As seen in Tables 4.3 and 4.4 the Row/Column approach was still able to solve the majority of LUTs on each FPGA, with the exception of the Cyclone V, which has the issue of LUT packing.

In another notable exception with this method, the Cyclone IV had issues with inconsistent placement in the the LUT Solving step. This meant that it required using the maximum hamming distance approach to pair LUT halves. Using just the exact inverse to match LUT halves yielded only 13819 solved LUTs in Run 1 and 13523 in Run 2. This was the only case where the exact inverse was not sufficient in solving all instantiated LUTs.

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14380	14380	29	14380	4
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	failed	failed	failed
Xilinx	Artix	xc7a100tfgg484-2L	63400	63384	63384	34	63384	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40992	40992	33	40992	6
Xilinx	Spartan	xc7s15cpga196-2	8000	8000	8000	25	8000	6
Xilinx	Spartan	xc7s100fgga484-2	64000	63992	63992	38	63992	6
Xilinx	Zynq	xc7z020clg484-1	53200	53192	53192	30	53192	6

Table 4.3: Row/Col Method: Run 1

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14370	14370	31	14370	4
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	failed	failed	failed
Xilinx	Artix	xc7a100tfgg484-2L	63400	63376	63376	32	63376	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40984	40984	33	40984	6
Xilinx	Spartan	xc7s15cpga196-2	8000	n/a	n/a	n/a	n/a	n/a
Xilinx	Spartan	xc7s100fgga484-2	64000	63984	63984	33	63984	6
Xilinx	Zynq	xc7z020clg484-1	53200	53184	53184	32	53184	6

Table 4.4: Row/Col: Run 2

Linear Snake The Linear Snake method performed slightly better than the Row/Column approach at getting maximimum single run LUT utilization. However, it also was not capable of solving the Cyclone V due to LUT packing.

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14399	14399	27	14399	4
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	failed	failed	failed
Xilinx	Artix	xc7a100tfgg484-2L	63400	63398	63398	34	63398	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40997	40997	32	40997	6
Xilinx	Spartan	xc7s15cpga196-2	8000	8000	8000	29	8000	6
Xilinx	Spartan	xc7s100fgga484-2	64000	63999	63999	32	63999	6
Xilinx	Zynq	xc7z020clg484-1	53200	53199	53199	32	53199	6

Table 4.5: Snake: Run 1

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14398	14398	29	14398	6
Intel	Cyclone V	5ceba4f17i7	18480	failed	failed	failed	failed	failed
Xilinx	Artix	xc7a100tfgg484-2L	63400	63397	63397	33	63397	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40995	40995	33	40995	6
Xilinx	Spartan	xc7s15cpga196-2	8000	n/a	n/a	n/a	n/a	n/a
Xilinx	Spartan	xc7s100fgga484-2	64000	63997	63997	32	63997	6
Xilinx	Zynq	xc7z020clg484-1	53200	53198	53198	31	53198	6

Table 4.6: Snake: Run 2

Modified Linear Snake The Modified Linear Snake method performed as well as the Snake method at maximizing single run LUT utilization while also solving the problem of LUT packing that caused the Row/Column and Snake methods to fail at solving the Cyclone V FPGA.

However, the Cyclone V once again posed an additional challange which we see in the number of bitstreams required to sort the LUT bits. The Cyclone V was the only FPGA that reordered the LUT bits with every input mask bitstream generated. This meant that obtaining N unique patterns for every LUT required a significant number of bitstreams. The problem is easy to imagine with an example. Say we have a 6 sided die, one side for each input of our 6 input LUT, and we want to observe each face at least one time. If we can pick-up and place the die then we only need to do so 6 times to observe every face. However, if we are forced to roll the die then it could take many more than 6 rolls to observe every face. This is what happens with the Cyclone V, and we do not just have one die, we have 18480.

This behavior of the Cyclone V was unexpected, because analysis of the XOR Mask indicated that there was no input pin reordering that occurred between the XOR and XNOR mask generation bitstreams, as there were very few noise bits in the mask. Furthermore, generating multiple bitstreams setting all LUTs to the same input mask value did not result in random changes in the input pin order, i.e. generating the same bitstream twice yielded two nearly identical bitstreams. This indicates that the Modified Snake pattern was successful of obtaining consistent placement, but there is some property if the input mask functions that results in changes to the input pin order. This meant that to solve the LUT bit order of the Cyclone V we had to generate multiple bitstreams in which each LUT was randomly

configured to one of the N input mask functions until we observed all N patterns in the bitstream for every single LUT.

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14399	14399	31	14399	4
Intel	Cyclone V	5ceba4f17i7	18480	18479	18479	28	18479	68
Xilinx	Artix	xc7a100tfgg484-2L	63400	63399	63399	33	63399	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40999	40999	35	40999	6
Xilinx	Spartan	xc7s15cpga196-2	8000	8000	8000	26	8000	6
Xilinx	Spartan	xc7s100fgga484-2	64000	63999	63999	32	63999	6
Xilinx	Zynq	xc7z020clg484-1	53200	53199	53199	31	53199	6

Table 4.7: Modified Snake: Run 1

Vendor	Family	Part	LUT Count	LUTs Utilized	LUTs Found	Number of Bitstreams	LUTs Sorted	Number of Bitstreams
Intel	Cyclone IV	ep4cgx15bf14a7	14400	14398	14398	29	14398	4
Intel	Cyclone V	5ceba4f17i7	18480	18478	18478	29	18478	62
Xilinx	Artix	xc7a100tfgg484-2L	63400	63398	63398	34	63398	6
Xilinx	Kintex	xc7k70tfbg484-3	41000	40995	40995	31	40995	6
Xilinx	Spartan	xc7s15cpga196-2	8000	n/a	n/a	n/a	n/a	n/a
Xilinx	Spartan	xc7s100fgga484-2	64000	63998	63998	31	63998	6
Xilinx	Zynq	xc7z020clg484-1	53200	53198	53198	32	53198	6

Table 4.8: Modified Snake: Run 2

4.5 Summary

In summary, we found that the data shows that the Modified-Snake LUT interconnect approach was the most effective interconnect pattern of the three, being able to solve the Cyclone V which the other two methods failed at. We demonstrated that Modified-Snake method was successful at consistently placing LUTs on the FPGAs that we tested. We also demonstrated that it is possible to reverse engineer the LUT programming bits by using generic VHDL while still maintaining reasonable efficiency in the number of bitstreams required.

Chapter 5

Conclusion

5.1 Future Work

This method could be improved by being formally verified or proved on physical hardware to test it's accuracy. We did not have access to physical versions of most of the FPGAs we tested on, rather relying on the consistency of the data to indicate that the mapping is accurate. Future work could also expand the concept of using VHDL to infer the instantiation of components to other portions of the FPGA like Block Ram (BRAM), Flip-Flops, Multiplexers. Modern compilers like Vivado and Quartus both support inferred BRAM. We have done some preliminary testing on REing BRAM bits using VHDL to infer and configure BRAM and it has shown promising results. Future work also includes attempting to reverse engineer the routing components such as Switch-Boxes to be able to reconstruct a netlist. Routing components pose a significantly more challenging problem than LUTs because using only VHDL there is little to no control over how the compiler chooses to route the design.

Reverse engineering the LUTs by themselves does not provide enough information to verify the integrity of a bitstream. However, it could be used for very primitive Trojan detection. For example, using the LUT mapping data one can determine the number of LUTs that are being utilized in a bitstream. FPGA compilers report the resource utilization of an FPGA after compilation, this includes the number of LUTs utilized. Using our methodology, one could compare the expected LUT utilization as reported by the tool with the actual LUT utilization on the resultant bitstream. Thereby providing a primitive way to check if any

unexpected logic was added directly to a bitstream. There could be some benefit in testing how effective such a strategy is at finding Trojans, though likely it would only work for a narrow subset of Trojans.

5.2 Closing Thoughts

While the methodology presented in this paper is limited in scope to targeting LUTs, the results show that it is possible to reverse engineer portions of the FPGA bitstream without needing to use specialty functions of a toolchain. The novel contributions of this methodology are the use of generic VHDL to instantiate and configure LUTs and the analysis that lets us overcome the challenges posed by input pin reordering and compiler optimizations. Our work shows that it is possible to reverse engineer portions of the bitstream format using generic VHDL which significantly reduces the effort required to implement the method on new FPGAs as compared to existing methods discussed in Chapter 2.

There probably will never be a truly generic way to reverse engineer all components on all FPGAs, but our research shows that it is possible to target ubiquitous components like LUTs with a generic methodology that can be applied to a wide variety of FPGAs. With additional research targeting other common FPGA components like BRAM, Flip-Flops, Multiplexers, and Switch-Boxes, it may be possible to have a generic method capable of reverse engineering the majority of an FPGAs bitstream with little upfront effort. Targeting these common FPGA components may allow for enough of a bitstreams netlist to be reconstructed to allow for the detection of some types of Trojans. Once the netlist or partial netlist is reconstructed, there is already significant research available on Trojan detection at the netlist level. It would be possible to use public tools like HAL [1] to do Trojan detection directly on the FPGA bitstream. There are existing methods [4] that have shown success in detecting FPGA Trojans by doing analysis on the gate-level netlists generated by the compiler, allowing for the detection of Trojans added at the design level. These existing methods could be applied to netlists extracted from the actual FPGA bitsreams, thereby also allowing for the detection of Trojans inserted directly into the bitstream.

Appendix A

A.1 Acronyms

ASIC - Application Specific Integrated Circuit

BRAM - Block Random-Access Memory

CB - Connector-Box

FPGA - Field Programmable Gate Array

HDL - Hardware Description Language

IP - Intellectual Property

LB - Logic Block

LUT - Look-Up-Table

RTL - Register Transfer Level

SB - Switch-Box

Appendix B

B.1 Quartus

VHDL parameter used to disable logic optimization on LUT interconnect net:

```
attribute keep : boolean;
attribute keep of NET : signal is true;
```

B.1.1 Quartus Build Script

```
# Software Used:
# Quartus Prime Design Software Version 20.1.0

project_new -revision logmap -overwrite logmap
set PART [lindex $argv 0]
set BITFILE [lindex $argv 1]
set_global_assignment -name DEVICE $PART
set_global_assignment -name VHDL_FILE LUT_Matrix.vhd
set_global_assignment -name VHDL_FILE LUT_case.vhd
set_global_assignment -name TOP_LEVEL_ENTITY LUT_Matrix
load_package flow
execute_module -tool map
execute_module -tool fit -args "--effort=standard"
```

```
execute_module -tool asm -args "--write_settings_file=on --read_settings_file=
exec cp logmap.sof $BITFILE
load_package report
load_report
puts [get_fitter_resource_usage -utilization]
project_close
```

B.2 Vivado

VHDL parameter used to disable logic optimization on LUT interconnect net:

```
    attribute dont_touch : string;
    attribute dont_touch of NET : signal is "true";
```

B.2.1 Vivado Build Script

```
# Software Used:
# Vivado v2020.2 (64-bit)

set PART [lindex $argv 0]
create_project -in_memory -part $PART
set_param general.maxThreads 6
read_vhdl -library xil_defaultlib {LUT_Matrix.vhd LUT_case.vhd}
synth_design -top LUT_Matrix -part $PART -flatten_hierarchy none
set_property SEVERITY {Warning} [get_drc_checks UCIO-1]
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
set_property BITSTREAM.GENERAL.CRC DISABLE [current_design]
place_design
route_design
write_bitstream -force [lindex $argv 1]
```

Bibliography

[1] EmSec Chair for Embedded Security. *HAL - The Hardware Analyzer*. https://github.com/emsec/hal. 2019.

[2] Louis-David Perron Etienne Bergeron and Marc Feeley. "Logarithmic-Time FPGA Bitstream Analysis: A Step Towards JIT Hardware Compilation". In: *ACM Transactions on Reconfigurable Technology and Systems* (2011). DOI: http://doi.acm.org/10.1145/1968502.1968503.

[3] A. Seffrin F. Benz and S. A. Huss. "Bil: A tool-chain for bitstream reverse-engineering". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)* (2012), pp. 735–738.

[4] Marc Fyrbiak et al. "Graph Similarity and its Applications to Hardware Security". In: *IEEE Transactions on Computers* 69.4 (2020), pp. 505–519. DOI: 10.1109/TC.2019.2953752.

[5] P. Perumalla H. Stowasser and J. M. Emmert. "An Abstract Approach to Algorithmic Time FPGA Reverse Engineering". In: *GOMACTech-22* (2022).

[6] Intel. *Cyclone V Device Handbook Volume 1: Device Interfaces and Integration.* https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/ug474_7Series_CLB.pdf. 2020.

[7] É. Note J.-B.; Rannaud. "From the bitstream to the netlist". In: *In Proceedings of the International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)* Volume 18, p. 264. (2008).

[8] *Project X-Ray*. 2018. URL: https://symbiflow.readthedocs.io/projects/prjxray/en/latest/.

[9] S. Guo T. Zhang J. Wang and Z. Chen. "A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code". In: *IEEE* vol. 7, pp. 38379-38389 (2019).

[10] Xilinx. *7 Series FPGAs CLB User Guide UG474 (v1.8)*. https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/ug474_7Series_CLB.pdf. 2016.