NAECON 2023 - IEEE National Aerospace and Electronics Conference | 979-8-3503-3878-2/23/$

Fast FPGA Reverse Engineering for Hardware Metering and Fingerprinting

Anvesh Perumalla
Dept. of ECE
University of Cincinnati
Cincinnati, OH, USA
perumaak@mail.uc.edu

Heiko Stowasser
Dept. of ECE
University of Cincinnati
Cincinnati, OH, USA
stowasho@mail.uc.edu

John M. Emmert
Dept. of ECE
University of Cincinnati
Cincinnati, OH, USA
john.emmert@uc.edu

***Abstract*—*We describe a fast, abstract method for reverse engineering (RE) field programmable gate array (FPGA) look-up-tables (LUTs). Our method has direct applications to hardware (HW) metering and FPGA fingerprinting, and our approach allows easy portability and application to most LUT based FPGAs. Unlike conventional RE methodologies that rely on vendor specific code (like Xilinx XDL), tools, configuration files, components, etc., our methodology is not dependent on any specific FPGA or FPGA computer aided design (CAD) tool. We use generic hardware description language (HDL) code based on specially connected CASE statements to program the LUTs on a target FPGA. Our specially connected CASE statements allow us to guide placement of LUT functions on successive synthesis runs. This enables us to quickly determine which bits in the FPGA's configuration file match to FPGA LUT bits. After we know which bits are LUT bits, we can go further and match specific LUT bits to specific bits in the configuration file, thereby creating a one-to-one mapping between every LUT memory cell and its matching bit in the configuration file. In this paper we present our CASE statement functions for performing one-to-one mapping of all FPGA LUT memory cell bits to specific configuration file bits. We have successfully applied our methods to several 7000 series Xilinx and Intel (Altera) FPGAs.*

Keywords*—*FPGA; Memometer; HW Metering; Reverse Engineering; Security; Assurance; Trust; Trojan

I. Introduction

Hardware (HW) metering helps with identification and locking of integrated circuits (ICs) that are manufactured under the same mask [1]. Passive HW metering is used to tag each IC with an unclonable, unique identifier. This identifier is further used in recognizing genuine ICs from overbuilt and counterfeit ICs. Whereas in active metering, in addition to tracking passively, it can also help with enabling/disabling IC functionality and controlling/preventing the ICs from further infiltrating the supply chain [1]. In our previous work, we developed Memometer, a low-overhead, inexpensive, adaptable hardware metering (fingerprinting) methodology that leverages memory physically unclonable functions (PUFs) [2-5]. In this work, we present a fast methodology for reverse engineering (RE) field programmable gate array (FPGA) programming files (or bit-files) to quickly determine look-up-table (LUT) programming bits [6].

This work was funded by the NSF Center for Hardware and Embedded Systems Security and Trust (CHEST) IUCRC through NSF Grant 1916722.

Standard RE of FPGA programming bit-files requires knowledge of how the bits in the bitstream map to the configurable logic for specific FPGAs. There are several methods to obtain this mapping. One notable effort is Project X-Ray, which targets 7 Series Xilinx FPGAs [7-8]. It should be noted that all the current methods of RE are limited either to a specific FPGA or to a specific manufacturer's toolchain [7-20].

This paper proposes an abstract method for obtaining the mapping of FPGA logic block (LB) LUT bits that is not tied to a specific FPGA or toolchain and can be used on any FPGA with LUT based Island-Style architecture, which encompasses most commercially available FPGAs [6]. The paper is organized as follows. First we describe specific advantages, claims, assumptions, and justifications we made. Then, we provide a detailed description of our abstract approach for RE FPGA configuration files to LUT bits and how it is applied to FPGA fingerprinting. Finally, we describe obstacles we overcame followed by our test, results, and analysis.

II. Advantages and Assumptions

The methodology proposed in this paper does not rely on manufacturer specific tools. It uses a high-level, HDL approach that leverages generic, but connected, CASE statements to instantiate LUT functions rather than using vendor specific components or macros. Additionally, rather than requiring placement to be specified, the connected CASE structures lead to consistent, repetitive LUT placement that rules out variations (primarily regarding routing strategies) between synthesis runs. To run the algorithm, only basic data from standard FPGA data books is required. The number and size (number of inputs) of FPGA LUTs is needed. In the case of more complex, decomposable LUTs, like those found in the Intel/Altera Cyclone V, some additional information about the structure (found in the FPGA data book) may also be needed.

Here we summarize our assumptions, followed by validation arguments. For our work we assume:

- Island style architecture with frame-based configuration memory (Fig. 1);
- We can synthesize **unencrypted**, FPGA configuration files with **no checksum** bits; and
- Synthesis tools allow **logic optimization** to be turned **off**.

Our first assumption is that the target FPGA has an island style architecture with frame-based configuration memory (Fig. 1). This is typical for most commercially available FPGAs, and even while most FPGAs also have other programmable logic structures (like built-in multipliers, clock-managers, phase-locked loops, RAM components, ARM processors, etc.) intermixed with the programmable logic LBs, we can still apply our algorithms to just the LB LUTs. As a side note, with slight modification, we have also successfully applied our algorithms to determine the configuration bits for other programmable logic structures, but this paper only describes the LUT configuration bits. For the second and third assumptions, most commercial CAD tools (like Xilinx Vivado and Intel/Altera Quartus) allow generation of configuration files with these options.

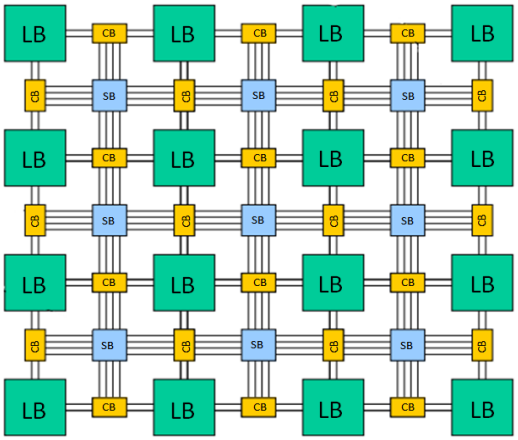

Fig 1: FPGA island style architecture and routing topology.

III. Methodology

In general, FPGA configuration files (or bitstreams) are loaded into FPGAs to program them. Our abstract methodology for RE FPGA bitstreams has two primary components, each with several parts.

- *Mapping the bits in the FPGA configuration bitstream to the programmable hardware in the FPGA.*
- *Obtaining a FPGA configuration file or bitstream to reverse engineer.*

The first component includes mapping the bits in the bitstream to the FPGA hardware used for both the programmable logic and programmable interconnect. The second component can be accomplished by A) synthesizing a configuration file using a commercially available CAD tool, B) obtaining the configuration bitstream from a design house or third-party vendor, C) reading the configuration file back from an already programmed FPGA or FPGA flash memory chip, or D) other method. Our other work, Memometer, can be used to readback configuration files from programmed FPGAs or FPGA flash memory chips [2-5].

The focus of this work is on the first component, mapping the bits in the FPGA bitstream to the FPGA programmable hardware, and more specifically, below is our abstract approach to mapping bitstream configuration bits to FPGA LUTs.

Our abstract approach to mapping bitstream bits to FPGA LUTs has two primary subtasks: A) creating a bitstream LUT ***mask***, and B) mapping individual bitstream bits to specific LUTs and LUT bits. The first subtask creates a ***mask*** or filter that identifies all LUT configuration bits in the bitstream. The second subtask assigns every LUT bit from the bitstream to a specific LUT configuration bit. The result of the two-step process is an unconnected ***graph*** where each node represents a FPGA LUT, and each node contains the value assigned to each of the LUT memory cells. For future work, we will map the programmable interconnect configuration bits in the bitstream to create a connected graph of the FPGA LUT functions.

A. Creating LUT *Mask*

The first subtask in our abstract approach is to determine all of the LUT configuration bits in the FPGA bitstream file or in other words, create a LUT ***mask***. We accomplish this using an abstract HDL program, and there are three pieces to this HDL program: i) a set of interconnected CASE statements that place LUT functions in consistent locations for multiple configuration file synthesis runs, ii) a set of CASE statement functions that we use to program LUT bits first to one logic value and then to its inverse for successive configuration file synthesis runs, and iii) a bit-wise XOR function that we use to create the actual LUT ***mask***. We will describe each below.

1) Consistent LUT placement algorithm, ***B***(N,K)

For the first part of our approach, we need to be able to consistently place an arbitrary function in a specific LUT location while we generate multiple FPGA configuration files. For simplicity, we describe the process for generating two FPGA configuration files, ***B*** and ***BI***, where all of the LUT bits in ***B*** are the inverse of the LUT bits in ***BI***. To accomplish this while not requiring any FPGA specific components or configuration files, we use a set of connected HDL CASE statements. There is one CASE statement for each LUT in the target FPGA, and by connecting the CASE statements to each other using special patterns, we (with a high degree of probability) ensure the desired function is mapped to the same LUT location for both ***B*** and ***BI***.

We developed both row/column based patterns as well as snake based patterns to connect the CASE statements in our HDL code. The snake pattern worked best, and it is presented here in Fig. 2. In Fig. 2, ***B***(*N*,*K*) represents the code used to generate FPGA configuration file ***B*** with *N*, *K* input LUTs. Configuration file ***BI*** is generated by replacing the function LUT with the function LUTn. The actual LUT functions (defined by assigning memory cell (MC) values) in Fig. 2 for generating ***B*** are described in subsubsection *2)* below. Fig.3 below shows how the CASE statement defined LUT functions are connected using a snake pattern.

2) Case state functions

In subsubsection *1)* above, we describe the inter-CASE statement connectivity used to drive consistent LUT placement for successive FPGA configuration file synthesis runs. In this section we describe the actual LUT functions used to program the memory cells (MCs) in ***B***(*N*,*K*). Our LUT functions are

based on XOR, XNOR, or Hamming functions of the LUT's input address bits. We use XOR, XNOR, or Hamming functions of the LUT's input address bits for two primary reasons: first they require use of all LUT inputs, so none of them are optimized away, and second, because the LUT inputs are not transposed during successive FPGA configuration file synthesis runs. Even with optimization turned off, other functions can have inputs optimized away (because they are not needed) or allow LUT input signals to be transposed (or moved to different LUT input locations).

```
Linear Snake MAP B (N, K):
        // N = # of K input LUTs on the FPGA
        // note: initial LUTs (LUT_j for j < K)
        //                    are handled as special cases
        // note: to generate BI,
        //                    replace all LUT() with LUTn()
        for j = K+1 to N
                LUT_j(LUT_j-K(), …, LUT_j-2(), LUT_j-1());
        end loop;
end;
Function: LUT(AK, …, A2, A1)
        // AK, …, A2, A1 = K address bits to address 2^K-1 MCs
        let A = binary2integer(AK, …, A2, A1);
        case A = 0: Out <= MC(0) value;
        case A = 1: Out <= MC(1) value;
        case A = 2: Out <= MC(2) value;
        …
        case A = 2^K-1: Out <= MC(2^K-1) value;
        return;
Function: LUTn(AK, …, A2, A1)
        Out <= NOT(LUT(AK, …, A2, A1));
```

Fig 2: Generic HDL code for generating consistent LUT placement in multiple FPGA configuration files ***B*** and ***BI***.

XOR and XNOR functions programmed into LUTs do not require additional explanation. However, we do further describe Hamming functions of the LUT input address bits. Simply stated, we define the Hamming weight of a MC in a LUT as the sum of the '1' values in its input address. Hamming functions of the LUT's input address bits are functions that place '1' (or conversely '0') only in LUT MCs whose input address bits have the same Hamming weight. For example, a K=4 input LUT has address bits, A = 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, or 1111. The address with Hamming weight of 0 is A= 0000. The addresses with Hamming weight of 1 are 0001, 0010, 0100, and 1000. The addresses with Hamming weight of 2 are 0011, 0101, 0110, 1001, 1010, and 1100. The addresses with Hamming weight of 3 are 0111, 1011, 1101, and 1110. The only address with Hamming weight of 4 is 1111. Given a K input LUT with even K, Hamming weight of K/2 works best.

To generate the inverse version of ***B***, ***BI***, we just invert the MC values described above.

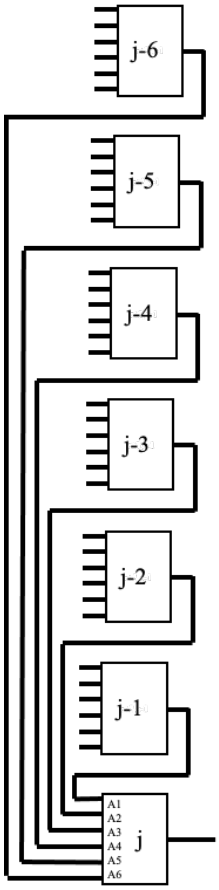

Fig 3: Snake pattern connections in ***B***(N,K) used to drive consistent placement of LUT locations for successive configuration file generation synthesis runs.

*3) XOR for **mask** creation*

The last step in **mask** creation is to XOR the ***B*** and ***BI*** configuration files. By leveraging the snake like connections between our interconnected LUT CASE statements (to drive consistent placement) and our XOR, XNOR, or Hamming functions (to define the MC values in the FPGA LUTs), we can create multiple FPGA configuration files where the only difference between the files is the values in the MCs. This allows us to do a bitwise XOR of the two files (***B*** and ***BI***) to create a **mask** that provides the location of each LUT in the FPGA configuration file. Using this **mask**, we further process to determine relative locations of LUT configuration bits.

B. Mapping Specific LUT MC Bits

Once we have the LUT **mask**, we use a combination of marching 1's and 0's, random functions, and a log based application algorithm to narrow down and determine specific LUT bits. With minimal human intervention, we successfully mapped all of the FPGAs we have tested. After applying our algorithms and functions, we have an unconnected graph with

relative (mapped to the bits in the configuration file) locations of all LUT programming bits. Our future work includes further processing the configuration file to include LUT connectivity in order to generate a connected graph of LUT functions for further processing and Trojan detection.

C. Fingerprinting FPGAs

Once we know which bits in the FPGA programming bit-file map to FPGA LUT memory cell bits, we can perform JTAG READBACK on the uncommitted FPGA LUT memory cells to determine their values at FPGA powerup [2-5]. The signatures provided by these memory cells can be applied to the Memory PUF used in Memometer to quickly and uniquely fingerprint the FPGA for HW metering purposes. More details on Memometer are found in [2-5].

IV. Obstacles

Before we describe our test results, we provide some detail on some of the obstacles we overcame in our approach described above.

A. Obstacle: Input Pin Reordering

For consistent placement of LUT functions, the connected CASE statement approach worked perfectly. However, as part of the place and route process, many compilers reorder the LUT input pin assignments to improve or reduce timing delays, even with logic optimization turned off. Input reordering changes the addresses of the bits used to program the LUTs, and if it occurs, interferes with LUT **mask** generation. In other words, the LUT bits in ***B*** and ***BI*** won't line up, so in that case, step *3) XOR for **mask** creation* does not always work. As described above, we overcame this obstacle by using Hamming functions of the LUT's input address. This solved the problem, and prevented LUT input address bits from being transposed during generation of ***B*** and ***BI***.

B. Obstacle: LUT Reduction via Input Elimination

Another obstacle we encountered was logic reduction due to input pin elimination. For example, even with logic optimization disabled, some compilers still perform optimizations on any LUT equations that can be reduced to fewer than K inputs. For example, the $K = 4$ input LUT function = "0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1" can be reduced to a function of just it's least significant address bit. Again, this interferes with **mask** creation since the LUT bits in ***B*** and ***BI*** may not line up. To overcome this in our approach described above, we made sure we only use functions (like XOR, XNOR, and Hamming) that require all input address bits.

C. Obstacle: LUT Merging

In our original implementation of the snake pattern described above, to initialize the start of the pattern (LUT_j for $j < K$), we used external inputs to drive the LUT inputs. This meant that initial LUTs next to each other in the chain shared all but one of their inputs. Some (but not all) compilers merged those LUTs into a single LUT. This was done by decomposing the shared inputs to drive several smaller LUTs whose outputs connected to multiplexers driven by the non-shared inputs to the two separate LUTs. Therefore, a different method was developed to connect the initial LUTs which guaranteed that none of the initial LUTs shared multiple inputs. This resolved the issue regardless of the synthesis compiler.

D. Obstacle: FPGAs That Leverage Partially Decomposable LUTs

While we were able to successfully use our approach on all FPGAs we explored, at least one family had issues. The Cyclone V from Intel/Altera uses a decomposable $K = 6$ input LUT structure that shares a common $K = 4$ input component. In other words, it is not completely decomposable. To overcome this, we had to slightly modify our generic code shown above to include multiple outputs from this $K = 6$ input LUT. The modification to the algorithm (shown in Fig. 2) was applied directly to all of the other FPGAs we tested with consistent, good results.

V. Test Results and Analysis

To test our approach we chose several Xilinx and Intel/Altera FPGAs. We directly applied the algorithm above to generate the **mask** files that define the LUT programming bits in the FPGA configuration files. The only time we slightly modified the algorithm was for the Cyclone V (as described in Obstacle IV.D above).

Table 1. Test Results for FPGA LUT Mask Generation

Vendor	***Family***	***Part***	***LUTs Available***	***LUTs Found***
Intel (Altera)	Cyclone IV	EP4CGX15BF14A7	14400	14400
Intel (Altera)	Cyclone V	5CEBAF17I7	18480	18480
Xilinx	Zynq	xc7z020clg484-1	53200	53200
Xilinx	Spartan	xc7s100fgga484-2	64000	64000
Xilinx	Artix	xc7a25tcpg238-3	14600	14600

Once the LUT **mask** was found for each device, it took O(log$N$) additional configurations to determine the location of each specific programming bit in the FPGA configuration file.

VI. Conclusions and Future Work

We have developed a generic, abstract approach to mapping FPGA configuration file bits to LUT programming bits. It is simple and the user is not required to be an FPGA expert. Our method does not require any vendor specific CAD tools, proprietary information, languages, configuration files or macros. Our work is directly applicable to FPGA HW metering and fingerprinting. Future work entails extending to Trojan detection for 3rd party IP.

REFERENCES

[1] Koushanfar, F., Hardware Metering: A Survey, eds. by M. Tehranipoor and C. Wang, *Springer* 2012.

[2] Perumalla, A.; Emmert, J.M. Memometer: Passive Memory-Based Metering System for Integrated Circuits. *GOMACTech-19*, 2019.

[3] Perumalla, A. and Emmert, J.M., Memometer: Memory PUF-Based Hardware Metering Methodology for FPGAs, *ACM Electronic Device Failure Analysis*, vol. 24, no. 2, November 2022.

[4] Perumalla, A. and Emmert, J. M., Memometer: Passive and Active Memory PUF-Based Hardware Metering Methodology for FPGA Supply Chain Security. *GOMACTech-23*, 2023.

[5] Emmert, J.M. and Perumalla, A., Memometer, Provisional US Patent 63/231,048.

[6] Emmert, J.M., Stowasser, H., Perumalla, A., Reverse Engineering Methodology for FPGA Bitstreams, Provisional US Patent 63,270,874.

[7] Bergeron, E.; Perron, L.D.; Feeley, M; David, J.P. Logarithmic-Time FPGA Bitstream Analysis: A Step Towards JIT Hardware Compilation. *TRETS*, 2011, 4.

[8] Project X-Ray. Available online: https://symbiflow.readthedocs.io/projects/prjxray/en/latest/ (accessed on 31 August 2021).

[9] Yu, H.; Lee, H.; Shin, Y.; Kim, Y. FPGA reverse engineering in Vivado design suite based on X-ray project. In Proceedings of the 2019 International SoC Design Conference (ISOCC), 2019, 239-240.

[10] Choi, S.; Park, J.; Yoo, H. Reverse Engineering for Xilinx FPGA Chips using ISE Design Tools. *J. Integr. Circuits Syst.* 2020, *6*, 1.

[11] Choi, S.; Yoo, H. Fast Logic Function Extraction of LUT from Bitstream in Xilinx FPGA. *Electronics* 2020, 9, 1132.

[12] Zhang, T.; Wang, J.; Guo, S.; Chen, Z. A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access* 2019, *7*, 38379–38389.

[13] Yu, H.; Lee, H.; Lee, S.; Kim, Y.; Lee, H.-M. Recent Advances in FPGA Reverse Engineering. *Electronics* 2018, *7*, 246.

[14] Yoon, J.; Seo, Y.; Jang, J.; Cho, M.; Kim, J.; Kim, H.; Kwon, T. A Bitstream Reverse Engineering Tool for FPGA Hardware Trojan Detection. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018.

[15] Seo, Y.; Yoon, J.; Jang, J.; Cho, M.; Kim, H.; Kwon, T. Poster: Towards reverse engineering FPGA bitstreams for hardware trojan detection. In Proceedings of the Network Distribution System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018.

[16] Jeong, M.; Lee, J.; Jung, E.; Kim, Y.H.; Cho, K. Extract LUT Logics from a Downloaded Bitstream Data in FPGA. In Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 27–30 May 2018.

[17] Wallat, S.; Fyrbiak, M.; Schlögel, M.; Paar, C. A Look at the Dark Side of Hardware Reverse Engineering—A Case Study. In Proceedings of the 2017 IEEE 2nd International Verification and Security Workshop (IVSW), Thessaloniki, Greece, 3–5 July 2017.

[18] Ding, Z.; Wu, Q.; Zhang, Y.; Zhu, L. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocess. Microsyst.* 2013, *37*, 299–312.

[19] Benz, F.; Seffrin, A.; Huss, S.A. Bil: A tool-chain for bitstream reverse-engineering. In Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, 29–31 August 2012.

[20] Note, J.B.; Rannaud, E. From the bitstream to the netlist. In Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, New York, NY, USA, 24–26 February 2008.