Malicious Trigger Discovery in FPGA Firmware

Timothy Dunham, Stephen Baka, John Hallman, Scott Harper
Secure Computing & Communications Division, MacAulay-Brown, Incorporated
Roanoke, VA, USA
fpga@macb.com

***Abstract*—FPGA netlists typically consist of hundreds to thousands of decoding structures that perform device operations. Malicious circuits (Trojans) often use complex decoders to delay activation until a time of an adversary's choosing. Identifying these triggers in a sea of legitimate decodings is highly desirable, but difficult to achieve; even with the assistance of a subject matter expert. This paper presents automation, developed on the DARPA VET program, that quickly discovers suspicious trigger circuits and verifies the correct operation for legitimate decoders.**

***Keywords*—FPGA; decoder; address; trigger; malice; Trojan; DARPA; vetting; CUD; firmware**

I. Introduction

When assessing Field Programmable Gate Array (FPGA) designs for deployment risk, one often encounters logic that decodes multi-bit values. This decoder logic is common in both standard device functionality and in attacker-inserted malice; particularly in their Trojan trigger mechanisms. The difficulty faced by an analyst is that there are often hundreds or thousands of decoders in the standard functionality, while any malice – if it exists – will only consist of a small number of decoders. The manual process of identifying these "needles in the haystack" of decoders is laborious and time-consuming.

This paper presents an automated method for detecting such decoders in and FPGA netlist and an expansion of that method to speed assessment of decoders across a netlist for potential exposure to malicious trigger mechanisms.

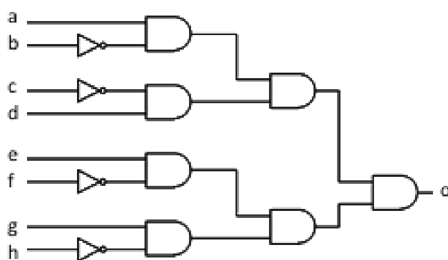

Fig. 1. Simple decoder. Output o = 1 when decoding 10011010 on inputs a:f

II. Background

The basic logic elements of FPGA designs are look-up tables (LUTs), flip-flops, multiplexers, and logic gates. These elements are combined to create nearly any digital circuit imaginable, from simple switching logic to complete system-on-chip (SoC) designs. Circuits that assert flags in response to specific multi-bit values are referred to as decoders. One way a system might use a decoder is to check for a value in its data stream. For example, an Ethernet router checks for a 16-bit EtherType field on incoming frames to enable specific packet handling logic. EtherType decoders would reduce the sixteen bits down to a set of 1-bit flags indicating specific values that are encountered in the traffic stream (e.g., 0x0800 for IPv4).

Decodings are also used to activate memory regions in response to specific address values. SoC designs often use a common bus fabric (e.g., AMBA AXI) that interconnects the various system components. An SoC slave (i.e., peripheral) only permits data access when an SoC master (e.g., CPU) initiates a transaction with the slave's address on the fabric. Detection of this address is performed with layers of decoders, where high-order address decoders enable access to the peripheral itself or its major functions, while low-order address bits select specific portions of the peripheral's memory space.

Finally, in the case of malicious insertions (i.e. Trojans) [1][2] into an FPGA design, an attacker would not want a Trojan to activate arbitrarily during normal device operation. Otherwise, the malice would easily be detected during product verification and removed. Thus the attacker wants to hide the malice behind a triggered event that is difficult to activate accidentally, but easily activated at a time of the adversary's choosing. Such malice activation triggers often require complex decoders that check for special data value sequences or make use of unused or reserved address spaces.

III. Method

The DARPA VET program developed methods for vetting commercial off-the-shelf (COTS) devices for vulnerabilities and determining their risk of use in DoD deployment. As part of this process, MacAulay-Brown (MacB) developed new techniques for vetting FPGA firmware in COTS devices for susceptibility to malicious incursion. Decoder analysis became part of MacB's vetting process [3] on the program. We found our analysts spent a lot of time manually decoding how buses were used in a system and thus developed an automated methods to quickly evaluate these buses. We then expanded the functionality to identify decoders and rank them by their likelihood of being a complex trigger circuit.

Here we discuss 1) our general approach used by our decoder analysis methods, 2) an expansion of those methods to discover decoders automatically, and 3) a method for ranking discovered decoders to indicate which might be suspicious.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and SPAWAR Systems Center Pacific (SSC Pacific) under Contract No. N66001-13-C-4045. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

A. General Method

The general approach is to discover which elements logically depend on a given starting point, and then evaluate the function of those elements to determine any decodings that include the starting point. To find the elements, our algorithm traces forward from the starting point until it reaches a prescribed stopping point. Originally, the stopping points included any complex components (i.e., not a LUT, gate, or flop) or system output pins. However, this approach resulted in masking of some of the decodings by subsequent logic.

A better approach was to stop tracing when the algorithm encountered a flop influenced by a combination of the user-provided starting point and some other data path outside the fan-cone of the starting point. This approach works in many cases, but there are scenarios where it would be useful for the algorithm to be greedier. A simple example where this operation failed occurred when the starting bus was immediately multiplexed with another bus and the output of the multiplexer was registered. The tracing stopped at the registers since the second bus originated outside the fan-cone of the user-selected starting point; stopping the algorithm before any practical decoding information had been gathered. Selecting useful stop points for the algorithm as it fans forward through the circuit is an ongoing area of research.

Once we have traced to a stopping point, we return to the user-specified starting point and generate Boolean logic equations for every component along the path traversed in the first pass. Some elements have simple Boolean equations, such as the basic AND and OR logic gates, but others such as LUTs are much more complex. As we move from element A to element B, we combine element A's equation into element B's equation by substituting equation A into equation B wherever the variable that represents element A occurs and then simplifying the result. If equation B becomes too large, in terms of processing time or number of inputs, we declare that it is performing complex operations and not simple decodings, and we stop analyzing that path. During this process, we keep track of whether the original starting bits have passed through flops, and treat these varying pipeline depths as different versions of the same input bit. All equation generation and analysis is performed with the CUDD [4] software package from Colorado University.

Once equation generation reaches the stopping points, the Boolean equations at those points are gathered and processed to determine the decodes of their inputs. Each minterm of the final equation is a potential decode. Any minterm that depends upon a propagated starting bit is treated as having decoded that bit. Different pipeline depths are reported individually. For ease of analyst understanding, decodes are reported in three formats: 1) for each stopping point, report all decodes; 2) for each decode, report stopping points at which it occurred; and 3) report which original inputs were used to specifically address recognized RAM structures. In addition, the user may indicate decodes that are expected to occur. If expected decodes are provided, an additional report section appears that describes where they occurred, any expected decodes that did not occur, and any discovered decodes that were not expected.

An issue with this approach is that the effort to decode a circuit region grows exponentially as more variables are added to a given Boolean equation. To ensure our method ran in a practical amount of time, the algorithm stopped processing a region if it took too long (>5 seconds per minterm) or ran low on memory. These regions were reported as error cases.

B. Global Tree Discovery

We found that many Xilinx decoders are implemented with a common structure where the output of a given element exclusively drives other elements previously identified as part of a decoder and no other logic outside the decoder. Additionally, these second-order elements exclusively source other elements in the decoder. This leads to a tree structure where the inputs being decoded are the leaves and the result of the decoding is the root. For example, Fig. 1 matches this structure: where each element has a single sink for its output and the tree converges upon a trunk at output *o*.

To quickly identify these tree structures, the algorithm creates a graph of the netlist where every component is represented by a node. It then adds edges to the graph only if they have a single sink. Finally, it locates all connected components in the graph and filters any components below a minimum size to obtain a list of component trees that are potential decoders. The minimum tree size and input width are user-selectable, but each defaults to two. We can further optimize results by constraining the algorithm's graph construction process. For example, decoders that contain registered components are less common, we can trim trees by omitting graph edges that are driven by a flop.

Once the list of trees is obtained, the algorithm then runs the general decoder analysis method on each tree, but restricts operation to just the elements in the tree. This informs us of the decodings performed within the tree. Unfortunately, there are hundreds or thousands of trees in a reasonably modern FPGA design, and we want to focus on only those that are actually decoders and likely trigger mechanisms for malice. To do this, we developed a scoring methodology for decodings that allows us to rank the trees by complexity.

C. Ranking Global Discoveries

To rank the discovered trees, we consider six factors representing common means by which an attacker would use a decoder to trigger malice. The factors are combined into a score and results in a sorted list of trees. Some factors make use of a tree's "largest decoding." Since some components in a tree can have complex equations, trees often have multiple decodings on which they could trigger. The "largest decoding" is simply the decoding in the tree that uses the most number of tree inputs. All factors (F_n) are normalized to real numbers in the range 0.0 to 1.0, then weighted and added together.

a) F_1: The percent of inputs used in the largest decode. A pure decoder will use all or most of its inputs as part of the decoding. If a large portion of the inputs is ignored, it is unlikely that the tree is a decoder.

b) F2: The width of the largest decode. Large decodes are more suspicious than smaller ones, because an attacker would want their trigger to be unique. The easiest way to ensure this is to use a large value. This also accounts for time-bomb triggers that are not just looking at a single large bit.

$$F_2 = 1 - \frac{1}{\sqrt{largest\ decode\ width\ (bits)}} \qquad (1)$$

c) F3: The ratio of input wires to input bits. In Verilog, nets can be a vector of multiple individual bits. When multiple bits are bused together, the analyst or some automated method has determined that they are used as a group instead of individually. If a tree is operating on a bus, then it is more likely to be decoding than if it were operating on a random selection of bits. The fewer input wires there are relative to the number of input bits, the more likely this tree is a decoder.

$$F_3 = 1 - \frac{N_{wires}}{N_{bits}} \quad (2)$$

d) F4: Whether the tree output drives a register Enable. A typical trigger event will be transient. An easy way to store a trigger event is to enable a flop for one cycle and permanently assert its output. For this factor, a simple 1 or 0 indicates whether or not the tree drives sequential Enable port.

e) F5: The number of decoded values in the tree. The more values that can cause the tree to change its output, the less likely it is that this tree is a decoder.

$$F_5 = \frac{1}{N_{decodes}} \quad (3)$$

f) F6: Number of other trees that share inputs with this tree. Some nets, such as address buses, are used to activate multiple regions of the netlist. These buses are decoded into many different values. They are also potential targets for malice because address spaces are rarely used completely. Undocumented, reserved, or unused addresses can be leveraged by an attacker to hide malicious triggers.

$$F_6 = 1 - \frac{1}{N_{trees}} \quad (4)$$

These six factors are then weighted and added together to form the final score *S* for the current tree. We tested various weights and found that we got the best results by evenly weighting all six factors. Thus the final tree score is to simply average the factors, as shown in Equation 5:

$$S = \frac{F_1+F_2+F_3+F_4+F_5+F_6}{6} \quad (5)$$

This process is applied to all trees in the netlist and the results are then sorted by score, in descending order. The first trees in the sorted list are the most likely to represent suspicious decodes. In practice, we have found that the first few trees have scores in the 0.7-0.9 range, and then they drop to under 0.5. The trees with scores under 0.5 are discarded, and the remaining trees are analyzed for potential malice.

IV. Evaluation and Results

Our decoder analysis methods were tested over the course of several "engagements" on the DARPA VET program that were designed to test our approach. Engagements used the NetFPGA [5] 10G board, a sandbox network card with four 10GigE SFP+ ports and a Gen2 x8 PCIe host connection, as an exemplar COTS product. An independent adversary team (i.e., USC ISI) created multiple FPGA designs per engagement to test our ability to both detect Trojans and avoid false alarms.

This paper focuses on Engagements 4 and 5; both of which implemented a network switch [6] based on the OpenFlow [7] specification. Engagement 5 FPGAs additionally included a simple encryption function for trusted traffic moving across untrusted network ports. In this paper, we will only discuss the contributions of our decoder analysis methods and not our results for the engagements as a whole.

A. First Evaluation

In Engagement 4 we did not have the global decoder analysis method, only the general method. Our approach was to manually identify locations in the netlist that were likely to be decoded, and check those decodings that were present. We were able to quickly rule out large sections of the netlist where there were either no decodings or only expected decodings. However, our analysts found that the most difficult aspect was running decoder analysis in the correct location. Of the ten malicious circuits inserted across the three OpenFlow Switch designs, three were well-suited to discovery via decoder analysis. However, only one of the three was found and here decoder analysis was used as a method of determining its function rather than a method of discovery.

a) Trigger after 31 packets of length 77 received. This malice had a decoder that checked the packet length field for a value of 77 and was not detected during the engagement.

b) Trigger on a special value in a packet. This malice triggered when a packet contained the 64-bit value 0x4d414c4943452036. This malice was detected through other methods and subsequently categorized with decoder analysis.

c) Trigger after 15 packets of length 83 received. This malice contained a decoder looking for packets of size 83, similar to the first malice listed above, but in a different article. This malicious circuit was not detected during the engagement.

The three malicious circuits listed above were only those with the potential to be caught via decoder analysis. While investigating our failure to detect two of those circuits during the engagement post-mortem, we created the global tree discovery algorithm. Once that method was complete, it immediately identified all three circuits. Circuit (a) was the highest scoring decoder tree in its article with a score of 0.63, and circuits (b) and (c) were the highest and second-highest ranked trees in another article with scores of 0.64 and 0.63, respectively. Thus, global tree discovery proved a promising method for discovering malicious triggers in FPGA firmware.

B. Second Evaluation

On Engagement 5 we used both the general decoding analysis method and the global tree discovery method. As with Engagement 4, the general method was largely used to rule out areas of the design where there were no unexpected decodings. However, the new tree discovery method pointed us directly at two malicious circuits, of the four possible circuits utilizing complex decoders, as trigger mechanisms. The tree method took approximately 30 minutes per FPGA netlist, running eight parallel operations of the general method at a time. Each of the three netlists contained approximately 158K components.

a) Trigger after receiving 8 packets of length 1149. This malice had a decoder that checked the packet length field for value 1149. The decoder was scored at 0.85 by the tree rankings. The trees with IDs 88871 and 84639 in Fig. 2 were the discoveries for this decoder.

b) Trigger on a special value in a packet. This malice triggered when a packet contained hex 0x4953494841564f43. It was in the same article as trigger (a) and the 3rd-ranked decoder with a score of 0.7465 (ID 83183 in Fig. 2).

c) Trigger after 101 packets match a condition. This malice counted packets that were forwarded from an outward-facing channel to the host. The counter to 101 was detected by tree discovery, but its score of 0.49 ranked lower than 200th in the list. We discovered this malice through other means.

d) Trigger after a sequence of 5 specific packet headers. This malice was triggered by a specific sequence of values seen over five packets. The trigger was crafted to evade tree detection and it succeeded. Even after accounting for evasion techniques in post-mortem, it only scored 0.38 (450th overall).

```
"id": 88871;
"output": "netlist.channel3_0.LUT6_51234_O";
"decode 0": "???_00010001_1111?11?";
"score": "0.851811";
--------------------------------
"id": 84639;
"output": "netlist.channel2_0.LUT6_51240_O";
"decode 0": "???_00010001_1111?1?1";
"score": "0.851811";
--------------------------------
"id": 83183;
"output": "netlist.channel2_0.LUT6_22944_O";
"decode 0":
"1_?1001001_01010011_01001001_01000100...";
"score": "0.746488";
--------------------------------
"id": 89116;
"output": "netlist.channel3_0.LUT6_18600_O";
"decode 0": "1_11111101";
"score": "0.733333";
--------------------------------
"id": 84776;
"output": "netlist.channel2_0.LUT6_54250_O";
"decode 0": "1_11111101";
"score": "0.733333";
```

Fig. 2. Top 5 tree discovery algorithm results for Article 3, Engagement 5.

That engagement demonstrated that while our tree discovery method significantly improved upon previous results, it was not a "silver bullet". There is still work to be done to make suspicious decoders rise in the rankings and the improve the identification of complex decoders in general. Fig. 2 illustrates that while the first three decoders ended up being malicious, there was not a big score differential with the benign decoders that immediately followed.

xparameters.h	Decoded Address Map
0x4060----	40_60_xx_xx
0x41a0----	*none found*
0x7a00----	7a_00_xx_xx
0x7a018---	7a_01_8x_xx
0x7d40----	7d_40_xx_xx

Fig. 3. Software headers vs decoded address map

One additional note for this engagement is that we were able to verify the address map of an internal addressable bus for the first time in an FPGA firmware vetting effort. The OpenFlow switch is built around AMBA AXI[8] and comes with PCIe software drivers. A subset of PCIe address bits are used internally as the peripheral selectors on the AXI bus. Fig. 3 shows the expected values from the driver software compared to the values decoded by our tool.

V. Future Work

As indicated by Engagement 5, there is room for improvement in both decoder detection and decoder ranking. For example, decoder detection could be automatically applied to all multi-bit output ports on large components; such as Block RAMs and digital signal processors. These components make convenient locations for an adversary to insert trigger mechanisms for malice and could be ranked in a similar fashion as the tree decoders to facilitate analysis.

An improvement to the general decoder analysis method would be a better determination of where to terminate checking for decodings. As discussed in §III.A, our initial stop point detection was too permissive while our current detection is sometimes too restrictive. Further research may find better heuristics for a middle ground.

VI. Conclusion

Automatic decoder analysis shows promise as a method for the vetting of FPGA-based commodity IT for exposure to malice. Quickly assessing regions of interest for a human analyst saves the time of manually deriving the combined decodes of that region. In addition, automated discovery of suspicious decoders (i.e., Trojan triggers) is effective as a way of focusing early FPGA firmware vetting efforts. There is still work to be done with many areas of approach, including the detecting the bounds of individual address regions, SoC address map reconstruction, and better heuristics for distinguishing malicious triggers from benign complex decoders across a given FPGA design.

VII. Acknowledgements

We thank DARPA I2O for their support in performing the research presented in this paper. We also thank the University of Southern California Information Sciences Institute (USC ISI) for providing the designs used to evaluate our methods.

VIII. References

[1] H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development" IEEE Int. Conference on Computer Design (ICCD), 2013.

[2] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, M. Tehranipoor, "Benchmarking of Hardware Trojans and Maliciously Affected Circuits", Journal of Hardware and Systems Security (HaSS), April 2017.

[3] S. Harper, J. Hallman, and S. Baka, "Vetting FPGA Firmware for Commodity Devices" GOMACTech 2016, March 2016.

[4] F. Somenzi, CUDD: CU decision diagram package - Release 3.0.0, Dep. Elect. Comput. Eng., Univ. Colorado, Dec 2015.

[5] Glen Gibb, John W. Lockwood, Jad Naous, Paul Hartke, and Nick McKeown. NetFPGA: An Open Platform for Teaching How to Build Gigabit-rate Network Switches and Routers. IEEE Transactions on Education, 2008.

[6] "OpenFlow implementation on NetFPGA-10G - Design Document," https://docs.google.com/document/d/1ZwHXQZocKwQls6Ted8VZO8h9MjBtu9WxV2fAY44eOgE/edit.

[7] "OpenFlow Switch Specification Version 1.0.0," http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf, December 2009.

[8] Xilinx, Inc., "AXI Reference Guide UG761," 2012.