Toggle MUX: How X-Optimism Can Lead to Malicious Hardware

ABSTRACT

To highlight a potential threat to hardware security, we propose a methodology to derive a trigger signal from the behavior of Verilog simulation models of field-programmable gate array (FPGA) primitives that behave X-optimistic. We demonstrate our methodology with an example trigger that is implemented using Xilinx 7 Series FPGAs. Experimental results show that it is easily possible to create a trigger signal that is '0' in simulation (pre- and post-synthesis), and '1' in hardware. We show that this kind of trigger is neither detectable by formal equivalence checks, nor by recent Trojan detection techniques. As a countermeasure, we propose to carefully reconsider the utilization of X-optimism in FPGA simulation models.

1. INTRODUCTION

Hardware Trojans have been recognized as a serious concern in the past decade. Numerous works exist to describe, classify and detect hardware Trojans at several abstraction levels. Recent surveys provide an overview of the field [9, 17]. In general, a hardware Trojan is a system that serves a shadow functionality besides its intended functionality, such as a backdoor or data leakage. In order to evade detection during functional tests, a hardware Trojan typically incorporates a trigger circuit which activates Trojan payload after the testing phase. In this work, we present such a trigger circuit. Although the trigger principally works for both application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) architectures, we focus on the latter. We exploit the X-optimistic behavior of Xilinx 7 Series FPGA simulation models to create a trigger circuit. Our trigger circuit generates a signal that is '0' during the design and simulation phase, and '1' when the design is implemented in real hardware.

1.1 Threat model

Our attack relies on a threat model, in which a malicious designer at an intellectual property (IP) vendor injects malicious functionality during the design process of an IP core. The designer obfuscates malicious functionality such that it is not obvious to code reviewers. This includes proper naming of signals and registers. During simulation, the compromised design behaves functionally equivalent to the original design specification. The resulting IP core is shipped to an IP integrator which incorporates the core in FPGA designs. The IP core is either shipped encrypted or unencrypted. If the IP core is encrypted, detection will be more difficult, because the IP integrator will be limited to a given set of tools which can process the encrypted IP core. As malicious functionality is added during design, the resulting functional specification (typically at register transfer level (RTL)) can not be trusted. Our circuitry extends the original design's functionality. In simulation, it does not violate its specification. We present a trigger that exploits common practice in FPGA simulation models with respect to X-propagation.

1.2 X-propagation

In integrated circuit (IC) design, functional simulation is used to verify the correctness of implemented functionality. During simulation, multi-value logic is used. Besides '0' and '1', the Verilog standard defines *unknown* ('X') and *tri-state* ('Z'). As the name suggests, a logic simulator assigns a signal the value 'X' if it cannot predict if it is '0' or '1', such as for uninitialized registers. When applied to the inputs of subsequent logic operations, such an 'X' can propagate through the design (e.g., '1' $\wedge$ 'X' = 'X', '0' $\vee$ 'X' = 'X', '0' $\oplus$ 'X' = 'X', '1' $\oplus$ 'X' = 'X'). This is commonly referred to as *X-propagation*. X-propagation can cause problems such as preventing a design to correctly reset in simulation. Therefore, certain Verilog constructs behave *X-optimistic*, which means that an 'X' is turned into a '0' or '1' [11]. X-optimism, however, can hide design bugs, especially at higher levels of abstraction (e.g., at the RTL). Therefore, *X-pessimism* could be a better strategy to verify the correctness of a design. X-pessimism means that an 'X' is passed to the output of a logic operation if it cannot be evaluated to either '0' or '1'.

1.3 Trigger process

In ASIC design, simulation models of logic operations and functional blocks tend to be X-pessimistic because an undetected bug could lead to exorbitant cost. ASIC design typically is associated with high cost, therefore highly-skilled verification teams are dedicated to identify the root cause of bugs when an 'X' is encountered during simulation. Things behave different in FPGA design. Typically, FPGA design houses are small to mid-sized companies that do not

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

have the resources to dedicate a verification team to design debugging. However, since X-propagation can lead to unintended behavior during design simulation, FPGA vendors tend to shape the simulation models of their library components to behave X-optimistic. As stated before, X-optimism can hide design bugs. We exploit this feature in order to derive a trigger signal from a multiplexer that is controlled by an ‘X’. We enforce the synthesis tool to use the simulation model of a multiplexer that outputs ‘0’ when it is controlled by an ‘X’. In hardware, there exists no ‘X’, which means that the multiplexer is finally controlled by ‘1’ or ‘0’. With few additional circuitry, we turn the multiplexer’s output to ‘1’. This way, we generate a trigger signal that is ‘0’ during design-time and ‘1’ in hardware.

2. RELATED WORK

Fern, Kulkarni, and Cheng propose a methodology to inject hardware Trojans into register transfer level (RTL) designs leveraging don’t cares [2]. As ‘X’ can mean both ‘0’ and ‘1’, their class of Trojans does not violate the system specification. In general, the class of Trojans presented in [2] is designed to leak data. This fundamentally differs from our proposed malicious circuitry, which implements an intelligent trigger in order to activate Trojan payload of any kind. The characteristic of our trigger signal is that it is ‘0’ throughout the simulation phase, leaving the compromised design functionally equivalent to the original specification. Once the design is implemented in hardware, the trigger signal turns to ‘1’.

Krieg, Wolf, and Jantsch recently proposed a methodology that can be used to produce a similar trigger signal for field-programmable gate array (FPGA) architectures [8]. The authors inject a malicious lookup table (LUT) into the hardware description language (HDL) specification of an intellectual property (IP) core using a compromised HDL frontend. In a second step, the LUT is reconfigured by the bitstream backend of a compromised place-and-route tool such that malicious functionality is activated. In hardware, malicious behavior is observable, while in the design and simulation phase it is not. This is similar to our approach. While their attack is sophisticated in automatically inserting malicious functionality in existing designs, the major drawback of their approach is the high effort required to mount the attack. The major advantage of this work compared to [8] is that we can generate a similar trigger signal without the need to compromise a design tool. Our trigger methodology only leverages common practice of X-optimism in Verilog FPGA simulation models. Therefore, an adversary’s only effort when mounting our attack is to inject specially crafted Verilog HDL code into an existing design.

3. METHODOLOGY

3.1 General

We present a methodology to create a trigger signal whose trigger condition is that the point in the design flow is reached when the design is implemented in target hardware. The compromised design is functionally equivalent to the original HDL specification. Our methodology is based on the standardized behavior of Verilog when a multiplexer is controlled by a signal whose value is ‘X’. In Verilog, a multiplexer can be inferred from two language constructs: either by a ternary operator (*:?*-operator), or by an *if/else*-statement. The Verilog standard [6] defines distinct behavior for the multiplexers generated from a ternary operator or an *if/else*-statement. If the conditional argument of the ternary operator carries an ‘X’ as its value, this means that the inferred multiplexer is controlled by an ‘X’. In this case, the multiplexer propagates an ‘X’ to its output when the inputs to the multiplexer carry distinct values (therefore behaving X-pessimistic). Things are different for a multiplexer that is inferred from an *if/else*-statement. If the condition in the if-statement carries an ‘X’ as its value, the Verilog standard defines that the *else*-branch must be evaluated.

We make use of this behavior in order to implement a trigger signal that is always ‘0’ during simulation, and ‘1’ after the design is implemented in hardware. Although our methodology is principally applicable to application-specific integrated circuit (ASIC) architectures, in this work we primarily focus on FPGA architectures. Figure 1b illustrates the behavior of such a signal. After the design is implemented in hardware, our trigger signal changes from ‘0’ to ‘1’.

All we need to create such a signal is to leverage the standard behavior of a multiplexer that is inferred from an *if/else*-statement and which is controlled by an ‘X’. In order to implement our trigger signal, we have to ensure that during simulation we provide a constant and reliable ‘X’-source to the multiplexer’s controlling input, and to provide a constant ‘0’-source to the multiplexer’s input that is inferred from the *else*-branch of the *if/else*-statement. If we can provide such constant sources of ‘X’ and ‘0’, we can ensure a constant ‘0’ at the output of the multiplexer during simulation.

Things become interesting, when this multiplexer is actually implemented in hardware, because the notion of ‘X’ only is defined for the simulation model of a design. If a signal’s value is ‘X’ in hardware, this means that it can either be ‘0’ or ‘1’. And it must be either ‘0’ or ‘1’. This is also true for the multiplexer’s control input. If so, the multiplexer propagates the corresponding input to its output. An abstract model of such a multiplexer is given in Figure 1c. However, we cannot determine the control input’s value in hardware a-priori. We must connect the multiplexer’s other input to a signal that is constant ‘1’, and make sure that the multiplexer propagates this ‘1’ to its output. One way to accomplish this is to toggle between the two inputs (i.e., between ‘0’ and ‘1’), and to delay the multiplexer’s output by one clock cycle. When OR’ing the original and the delayed versions of the multiplexer’s output, the result is a constant ‘1’. In order to reflect its toggle behavior, we call this concept *Toggle MUX*. A high-level model of the trigger signal generator incorporating a Toggle MUX is given in Figure 1d. A more detailed schematic of the trigger generator is provided in Figure 2. Listing 1 provides the Verilog HDL specification of our trigger signal generator.

In the following, we provide details on how we generate signals whose values are constant ‘0’, ‘1’, and ‘X’. Also, we describe how we enforce the design tool to instantiate a multiplexer that implements the behavior we need to generate our trigger signal.

3.2 ‘X’-generation

In order to provide the control input of the Toggle MUX with a signal that constantly and reliably carries an ‘X’ value

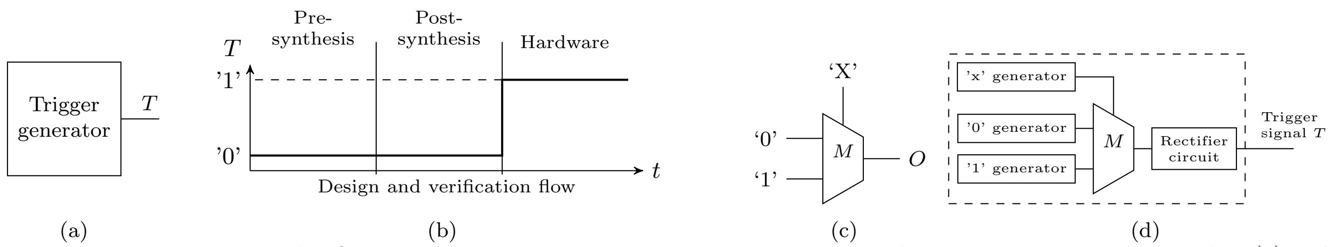

Figure 1: Function principle of our trigger. We inject a trigger generator circuit that generates a trigger signal T (a). The trigger signal is constant '0' during design time and becomes constant '1' when implemented in hardware (b). The core element of our trigger signal generator is a multiplexer which is controlled by a signal which is 'X' during design time. In simulation, output O of the multiplexer is determined by the *else*-branch of the inferring *if/else*-statement (X-optimistic behavior) (c). By wisely crafting Verilog HDL, the multiplexer can be forced to help in generating a trigger signal as shown in (b). A high-level schematic of the trigger signal generator is shown in (d).

during simulation, we generate such a signal that is persistent even if a global reset is applied to the design. We achieve such behavior by using a digital signal processor (DSP) core available on the target FPGA (Lines 23 to 30 in Listing 1). Making our trigger implementation depending on a DSP core may seem limiting. However, in order to demonstrate the feasibility of our attack, we think that the use of a DSP core is acceptable. We configure the DSP core in a way that it outputs an 'X' at its `OVERFLOW` output (Listing 7). In principle, this is the only output we need to generate an 'X'. However, using only the `OVERFLOW` output might appear suspicious during design review, therefore we also use the DSP core's data output (which we configure to be constant). In order to preserve the 'X', we combine the multi-bit data output and the `OVERFLOW` output by an XOR operation (because $A \oplus$ 'X' = 'X'). In a real-world design, it is very likely that a DSP core is used. In this case, the existing DSP core can be used to also generate the 'X' signal.

In hardware, the DSP core's output will actually be '0' or '1'. It depends on the target architecture which value an 'X' will be in hardware. Because we cannot predict this value, we add a toggling signal as input to the XOR operation (Line 11 and Line 20 in Listing 1). This way, during simulation, the XOR's output is still 'X'. However, in hardware the XOR's output toggles with each clock cycle. This way, we are able to guarantee that a '1' can propagate to the multiplexer's output O, which also toggles in hardware. In order to generate a constant '1' as specified in Figure 1b, we shape O with a digital rectifier, whose output T is constant '1'.

In a hand-optimized real-world design, it is usual that a DSP core is manually instantiated, therefore not being suspicious to HDL analysis. Once there is a DSP core present in a design, we can adapt its configuration to perform additional functionality to derive an 'X'. However, there may be other options to create an 'X'.

3.3 '0'/'1'-generation

As already stated, the Toggle MUX has to be provided a constant '0' to its input that is inferred from the *else*-branch, and a constant '1' to the input that is inferred from the *if*-branch of the *if/else*-statement. In order to prevent optimization passes to optimize away these constant signals, we use linear-feedback shift registers (LFSRs) to derive such signals. We use different polynomials in order to prevent the optimizer to merge the LFSRs for '0' and '1' generation (Line 8, Line 12, and Line 13 in Listing 1). An LFSR will never carry all register bits '0'. In order to derive a '0', we compare if the LFSR is equal to '0' (Line 32 in Listing 1). Likewise, we compare if the LFSR is unequal to '0' to generate a '1' (Line 33 in Listing 1). The outputs of the comparators C_1 and C_2 in Figure 2 carry the constant '0' and '1' signals.

3.4 Toggle MUX

The core part of our trigger is a multiplexer that is inferred from a Verilog *if/else*-statement, and which is controlled by an 'X' (Lines 38 to 41). It is important that this behavior persists over the entire design and simulation phase. We therefore need a multiplexer, which again uses an *if/else*-statement in its Verilog simulation model (and not the ternary operator). We investigated the simulation models for Xilinx FPGAs and found out that the `MUXF7` primitive satisfies this requirement. It is unusual to manually instantiate a `MUXF7` primitive, which would appear suspicious during design review. Instead of manual instantiation, we use a coding style that forces the synthesis tool to instantiate a `MUXF7` cell.

Xilinx 7 Series FPGA architectures provide dedicated multiplexers to combine the outputs of LUTs [1]. A `MUXF7` primitive combines the outputs of two 6-input LUTs (which are both part of the same logic cell of the FPGA). This means, that in order to enforce the synthesis tool to instantiate a `MUXF7`, we have to make sure that the *if/else*-statement in Listing 1 performs logic functions that can be mapped to 6-input LUTs. In our case, these logic functions are the comparators C_1 and C_2 in Figure 2, or the generation of `signal_0` and `signal_1` in Listing 1, Lines 32 to 33. Comparators C_1 and C_2 directly process the outputs of the respective LFSRs without any logic in between. Therefore, we use 6-bit wide LFSRs to enforce C_1 and C_2 to be mapped to 6-input LUTs. As a result, the *if/else*-statement is mapped to a `MUXF7`, and we can exploit its simulation behavior when it is controlled by an 'X'.

3.5 Rectifier circuit

When simulated, the Toggle MUX constantly outputs '0' (as illustrated in Figure 1b). Once the design is implemented in hardware, output O of the MUX toggles. In order to generate a constant '1', we need to *rectify* O. We achieve this task by delaying O by one clock cycle (Line 42) and OR'ing it with the original O (Line 45). This way, a trigger signal T is generated that constantly outputs '1'.

4. DEMONSTRATION

We simulate our trigger using Xilinx Vivado®, Version 2016.3, and experimentally evaluate our design on a Basys™ 3 FPGA board. Using Xilinx Vivado®, we perform simulations of our trigger at several stages in the design flow: 1. behavioral simulation, 2. post-synthesis simulation (including timing), and 3. post-implementation simulation (including timing). At any stage, the trigger shows expected behavior, i.e., propagating the DSP's 'X' output value such that trigger output T is '0' during simulation.

When we map the design to the bitstream and configure the target FPGA, the design again shows intended behavior. Now, 'X' is set to either '0' or '1', which forces trigger output T to be '1' (which is indicated by turning on a light emitting diode (LED) on the evaluation board). Listings 2 to 7 provide the complete code in order to reproduce our trigger circuit. The schematic of our trigger circuit is shown in Figure 2. The timing diagram of our trigger circuit is shown in Figure 3.

Listing 1: Verilog model of the trigger circuit (`trigger.v`). For the sake of clarity, we print only the relevant parts of the instantiated DSP core. The entire listing of the DSP core can be found in Listing 7

```
`timescale 1 ns / 1 ps

module trigger (
        input clk,
        output trigger
);
        reg toggle;
        reg [5:0] lfsr_0 = 1, lfsr_1 = 2;

        always @(posedge clk) begin
                toggle <= !toggle;
                lfsr_0 <= {lfsr_0, lfsr_0[5] ^
                     lfsr_0[4]};
                lfsr_1 <= {lfsr_1, lfsr_1[5] ^
                     lfsr_1[4]};
        end

        wire [48:0] signal_x_vec;
        reg signal_x;

        always @(posedge clk) begin
                signal_x <= ^{signal_x_vec, toggle};
        end

        DSP48E1 #(
                [...]
        ) signal_x_dsp (
                [...]
                .OVERFLOW(signal_x_vec[48]),
                .P(signal_x_vec[47:0]),
                [...]
        );

        wire signal_0 = lfsr_0 == 0;
        wire signal_1 = lfsr_1 != 0;

        reg trigger_a, trigger_b;

        always @(posedge clk) begin
                if (signal_x)
                        trigger_a <= signal_1;
                else
                        trigger_a <= signal_0;
                trigger_b <= trigger_a;
        end

        assign trigger = trigger_a || trigger_b;
endmodule
```

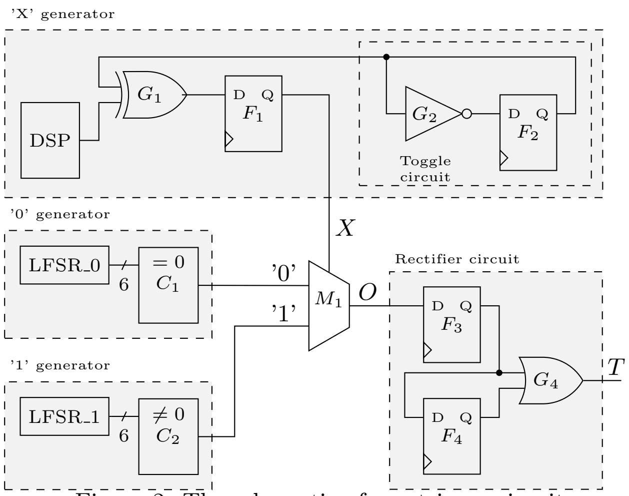

Figure 2: The schematic of our trigger circuit

5. DISCUSSION

X-Analysis tools that aim at detecting 'X's primarily target X-induced functional errors that propagate to the primary outputs of a system. However, because the multiplexer we use in our trigger circuit behaves X-optimistic, it prevents an 'X' at its control input to propagate. Also, the 'X' generated in our trigger circuit is persistent even when a global reset is applied to the system and will therefore not render our trigger useless. Simply simulating the system and manually analyzing all occurring 'X's will not be feasible, as the number of 'X's during design simulation is considered prohibitively high. Because the system lacks a golden reference, a formal equivalence check between the hardware representation and the original HDL will fail in detecting our trigger. Since our trigger is always on in hardware, detectability of malicious functionality by in-circuit testing highly depends on the details of Trojan payload implementation. In the following, we qualitatively evaluate the detectability of our trigger against the recent design-level detection approaches unused circuit identification (UCI) [4], functional analysis for nearly-unused circuit identification (FANCI) [15], functional identification of gate-level hardware trustworthiness (FIGHT) [10], Trojan prevention and detection (TPAD) [16], gate-level information flow tracking (GLIFT) [12] and Verifiable ASICs [14].

UCI is a dynamic approach proposed by Hicks et al. that analyzes the data flow graph (DFG) of a given design [4]. UCI flags input signals suspicious that do not impact (direct and indirect) output signals, which are therefore considered unused. Our proposed trigger will be detected by UCI for the case when 'X' is propagated to the control input of multiplexer M_1. However, according to [3], UCI only supports

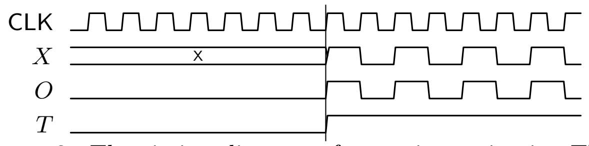

Figure 3: The timing diagram of our trigger circuit. The vertical line indicates the point in time when the design is implemented in hardware (and therefore the trigger condition is satisfied)

two-valued logic, considering both cases for ‘X’-valued signals during simulation, therefore setting such signals ‘0’ and ‘1’. As the DSP’s output will be set to either ‘0’ or ‘1’, the output of XOR G_1 will toggle, and therefore also multiplexer M_1 will toggle. This means that the circuit will not be unused and therefore not be detected by UCI.

FANCI is a static approach that aims to find rarely used signals in a gate-level netlist by analyzing the truth tables of all output signals’ fan-in trees. As Sullivan et al. point out, FANCI fails in correctly evaluating sequential blocks and feedback loops [10]. Thus, FANCI will not detect the constant ‘0’ and ‘1’ signals, which are generated using LFSRs. [15] does not indicate explicit consideration of ‘X’ inputs. Therefore, FANCI will not flag output O suspicious, because the control input X will toggle (which results in a toggling output signal O). Because FANCI treats sequential logic as non-clocked combinational blocks, neither T will be flagged suspicious by simply evaluating the truth table of the circuit (because the truth table does not represent any notion of time).

FIGHT is an extension to FANCI proposed by Sullivan et al. which enables the evaluation of designs that contain sequential logic and feedback loops [10]. While FIGHT allows to compare IP cores with similar functionality provided by different vendors, it does not support to detect single rare signals [7]. Therefore, our trigger will remain undetected. It will, however, depend on whether the triggered Trojan payload will propagate rare signals. If so FIGHT will probably flag the design suspicious due to such rare signals.

Wu et al. present TPAD, which is an approach to detect hardware Trojans during design-time testing and post-deployment [16]. In their methodology, a checker module is derived and synthesized from a given high-level specification of the design under test. TPAD addresses hardware Trojans that result from a malicious manufacturer and/or malicious design tools that inject extra functionality. TPAD requires a trusted version of the RTL or system specification. Our threat model, however, assumes a malicious designer injecting and obfuscating additional RTL. Therefore, the RTL specification cannot be trusted and TPAD will fail in revealing our trigger. Also, TPAD aims at detecting systems that behave incorrectly at runtime. Our trigger is fully specified at design-time, showing a subset of specified behavior during design-time (i.e., producing a constant ‘0’), and the complementary behavior at runtime (i.e., producing a constant ‘1’). At no point, our trigger behaves out of specification.

Wahby et al. present the *Verifiable ASICs* approach to verify that hardware systems operate correctly [14]. The authors propose to leverage *technology gap* in order to justify trust in a verifier core. In principle, the Verifiable ASICs approach implements a challenge response protocol which verifies that the result of a computation is correct. According to [13], the Verifiable ASICs approach could probably detect the trigger presented in this paper. However, it would be very impractical to leverage technology gap for FPGA systems. Nevertheless, adversaries potentially could modify design tools for 20-year-old FPGA technology, therefore greatly reducing trust imposed by technology gap.

Another promising approach to detect malicious hardware proposed by Tiwari et al., GLIFT, is to track the flow of information in order to detect data leakage caused by hardware Trojans [12]. Hu et al. study the impreciseness problem of GLIFT on the example of a 2-input multiplexer for the theoretical case if the select input of the multiplexer is both ‘0’ and ‘1’ at the same time. This is a condition that could be considered similar to the condition when the select signal in our Toggle MUX is ‘X’. In any case, GLIFT is effective to detect two types of Trojans: Those which leak data, and those which alter register values. Therefore, our trigger alone will not be detected by GLIFT, detection will instead depend on the Trojan payload activated by our trigger.

6. CONCLUSIONS

We presented an attack that leverages common practice in Verilog simulation models with regard to X-propagation. As discussed, recent methods to detect hardware Trojans will not reveal our trigger, because: (1) When the synthesized hardware is verified, no golden reference exists against which a design under verification could be checked because the original RTL is compromised. (2) When the simulation model of the design is checked, there is no unused or rarely used signal or circuit which could be detected. An effective countermeasure would be to allow X-propagation in FPGA design and verification. This would cause our Toggle MUX to propagate an ‘X’ instead of forcing its output to ‘0’, thus disabling its usage as a trigger. However, X-propagation will increase the number of ‘X’ in simulation results, requiring design engineers to thoroughly verify their designs. FPGA design requires less resources compared to ASIC design. FPGA design houses therefore typically do not have dedicated teams for hardware verification. This is why FPGA vendors make huge efforts to prevent X-propagation in order to satisfy customers (which do not have to care about ‘X’s in their simulation results). Therefore, allowing X-propagation would be a trade-off between correct hardware behavior and ease of verification.

Acknowledgments

This line should hide affiliation information.

References

[1] *7 Series FPGAs Configurable Logic Block User Guide*. Tech. rep. Xilinx, Inc., Sept. 27, 2016.

[2] N. Fern, S. Kulkarni, and K. T. T. Cheng. “Hardware Trojans hidden in RTL don’t cares — Automated insertion and prevention methodologies”. In: *Test Conference (ITC), 2015 IEEE International*. 2015, pp. 1–8.

[3] M. Hicks. *Personal E-Mail Communication on How UCI treats 'X' input signals*. Nov. 18, 2016.

[4] M. Hicks et al. “Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically”. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. May 2010, pp. 159 –172.

[5] W. Hu et al. “Theoretical Fundamentals of Gate Level Information Flow Tracking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (2011), pp. 1128–1140.

[6] “IEEE Standard Verilog Hardware Description Language”. In: *IEEE Std 1364-2001* (2001), pp. 1–856.

[7] Y. Jin. *Personal E-Mail communication if FIGHT detects single unused signals*. Nov. 19, 2016.

[8] C. Krieg, C. Wolf, and A. Jantsch. “Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow”. In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD ’16. Austin, Texas: ACM, 2016, 43:1–43:8.

[9] H. Li, Q. Liu, and J. Zhang. “A survey of hardware Trojan threat and defense”. In: *Integration, the {VLSI} Journal* 55 (2016), pp. 426 –437.

[10] D. Sullivan et al. “FIGHT-Metric: Functional Identification of Gate-Level Hardware Trustworthiness”. In: *Proceedings of the 51st Annual Design Automation Conference*. DAC ’14. San Francisco, CA, USA: ACM, 2014, 173:1–173:4.

[11] S. Sutherland. “I’m Still In Love With My X!” In: *Proceedings of the Design and Verification Conference (DVCon)*. 2013.

[12] M. Tiwari et al. “Complete Information Flow Tracking from the Gates Up”. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 109–120.

[13] R. S. Wahby. *Personal conversation regarding detectability of Toggle MUX by Verifiable ASICs approach, and the appicability of Verificable ASICs to the detection of Toggle MUX.* Nov. 14, 2016.

[14] R. S. Wahby et al. “Verifiable ASICs”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 759–778.

[15] A. Waksman, M. Suozzo, and S. Sethumadhavan. “FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis”. In: *Proceedings of CCS 2013*. Authors version. To be published in the Proceedings of the CCS 2013. 2013.

[16] T. F. Wu et al. “TPAD: Hardware Trojan Prevention and Detection for Trusted Integrated Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.4 (2016), pp. 521–534.

[17] K. Xiao et al. “Hardware Trojans: Lessons Learned After One Decade of Research”. In: *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (May 2016), 6:1–6:23.

APPENDIX

Listing 2: Testbench (`testbench.v`)

```
`timescale 1 ns / 1 ps

module testbench;
        reg clk = 1;
        wire trigger;

        always #50 clk = ~clk;

        trigger uut (
                .clk(clk),
                .trigger(trigger)
        );

        initial begin
                // $dumpfile("testbench.vcd");
                // $dumpvars(0, testbench);
                repeat (1000) @(posedge clk);
                $finish;
        end

        always @(posedge clk) begin
                $display("%t_%b", $time, trigger);
        end
endmodule
```

Listing 3: Constraints file (`trigger.xdc`)

```
set_property -dict { IOSTANDARD LVCMOS33 PACKAGE_PIN W5  } [get_ports clk ]
set_property -dict { IOSTANDARD LVCMOS33 PACKAGE_PIN U16 } [get_ports trigger]
```

Listing 4: Synthesis script (`trigger.tcl`)

```
create_project -part xc7a35tcpg236-1 -force vivadoprj

read_verilog trigger.v
read_verilog testbench.v
read_xdc trigger.xdc

synth_design -top trigger

opt_design
place_design
route_design

write_verilog -force -mode timesim trigger_post.v
write_bitstream -force trigger.bit
```

Listing 5: Simulation script (`runsim.sh`)

```
#!/bin/bash

set -ex

xvlog testbench.v
xvlog trigger.v
xelab --runall -L unisims_ver testbench work.glbl

xvlog trigger_post.v
xelab --runall -L unisims_ver testbench work.glbl
```

Listing 6: FPGA configuration script (`program.tcl`)

```
open_hw
connect_hw_server
open_hw_target [lindex [get_hw_targets] 0]
set_property PROGRAM.FILE trigger.bit [lindex [get_hw_devices] 0]
program_hw_devices [lindex [get_hw_devices] 0]
```

Listing 7: Full instantiation of DSP core in Listing 1

```
        DSP48E1 #(
                .ACASCREG(0),
                .ADREG(1),
                .A_INPUT("DIRECT"),
                .ALUMODEREG(0),
                .AREG(0),
                .AUTORESET_PATDET("NO_RESET"),
                .BCASCREG(0),
                .B_INPUT("DIRECT"),
                .BREG(0),
                .CARRYINREG(0),
                .CARRYINSELREG(0),
                .CREG(1),
                .DREG(1),
                .INMODEREG(0),
                .MASK(48'h3FFFFFFFFFFF),
                .MREG(1),
                .OPMODEREG(0),
                .PATTERN(48'h000000000000),
                .PREG(0),
                .SEL_MASK("MASK"),
                .SEL_PATTERN("PATTERN"),
                .USE_DPORT("FALSE"),
                .USE_MULT("MULTIPLY"),
                .USE_PATTERN_DETECT("NO_PATDET"),
                .USE_SIMD("ONE48")
        ) signal_x_dsp (
                .A({5{lfsr_0}}),
                .ACIN(0),
                .ACOUT(),
                .ALUMODE(0),
                .B({3{lfsr_1}}),
                .BCIN(0),
                .BCOUT(),
                .C(1),
                .CARRYCASCIN(0),
                .CARRYCASCOUT(),
                .CARRYIN(0),
                .CARRYINSEL(0),
                .CARRYOUT(),
                .CEA1(0),
                .CEA2(0),
                .CEAD(0),
                .CEALUMODE(0),
                .CEB1(0),
                .CEB2(0),
                .CEC(0),
                .CECARRYIN(0),
                .CECTRL(0),
                .CED(0),
                .CEINMODE(0),
                .CEM(0),
                .CEP(0),
                .CLK(clk),
                .D(0),
                .INMODE(0),
                .MULTSIGNIN(0),
                .MULTSIGNOUT(),
                .OPMODE(0),
                .OVERFLOW(signal_x_vec[48]),
                .PATTERNBDETECT(),
                .PATTERNDETECT(),
                .PCIN(0),
                .PCOUT(),
                .P(signal_x_vec[47:0]),
                .RSTA(0),
                .RSTALLCARRYIN(0),
                .RSTALUMODE(0),
                .RSTB(0),
                .RSTC(0),
                .RSTCTRL(0),
                .RSTD(0),
                .RSTINMODE(0),
                .RSTM(0),
                .RSTP(0),
                .UNDERFLOW()
        );
```