# Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow

Christian Krieg
Institute of Telecommunications, TU Wien,
Gusshausstr. 25 / 389, 1040 Wien, Austria
christian.krieg@alumni.tuwien.ac.at

Clifford Wolf and Axel Jantsch
Institute of Computer Technology, TU Wien,
Gusshausstr. 27–29 / 384, 1040 Wien, Austria
clifford@clifford.at, axel.jantsch@tuwien.ac.at

## ABSTRACT

We present a novel type of Trojan trigger targeted at the field-programmable gate array (FPGA) design flow. Traditional triggers base on rare events, such as rare values or sequences. While in most cases these trigger circuits are able to hide a Trojan attack, exhaustive functional simulation and testing will reveal the Trojan due to violation of the specification. Our trigger behaves functionally and formally equivalent to the hardware description language (HDL) specification throughout the entire FPGA design flow, until the design is written by the place-and-route tool as bitstream configuration file . From then, Trojan payload is always on. We implement the trigger signal using a 4-input lookup table (LUT), each of the inputs connecting to the same signal. This lets us directly address the least significant bit (LSB) and most significant bit (MSB) of the LUT. With the remaining 14 bits, we realize a "magic" unary operation. This way, we are able to implement 16 different Triggers. We demonstrate the attack with a simple example and discuss the effectiveness of the recent detection techniques unused circuit identification (UCI), functional analysis for nearly-unused circuit identification (FANCI) and VeriTrust in order to reveal our trigger.

## 1. INTRODUCTION

Hardware Trojans became a major concern during the last years. Several threat models, attacks and countermeasures have been described at any stage of the design flow for multiple technologies such as application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA). A hardware Trojan is an electronic system that serves a shadow purpose besides its intended functionality. The hidden functionality is unspecified and undocumented, and included in a malicious manner. In order to evade detection during functional tests, a hardware Trojan typically implements a trigger mechanism which activates a payload mechanism such that Trojan payload is available in the field. A multitude of different trigger implementations have been published so far, the most of them relying on rare events, such as the occurrence of patterns on a bus or the elapse of a time interval bigger than the time to finish functional testing. Also, physical triggers have been proposed. [3, 15]

State-of-the-art techniques to detect hardware Trojans at design time exploit the fact that they are activated by triggers, therefore during design time there are portions of the circuit which are rarely used, which consequently are to be identified in order to detect malicious functionality [5, 18, 22]. In this work, we present a trigger mechanism that cannot be detected by recent approaches to design-time hardware Trojan detection. We attack the design flow at two stages: In a first step, when the hardware description language (HDL) representation of the design is read by the synthesis tool, we inject malicious functionality that incorporates camouflaging circuitry which prevents malicious functionality from being detected. In this stage, the compromised design behaves functionally and formally equivalent to the original HDL. Also, any signal of the malicious functionality is used, which makes state-of-the-art approaches unable to detect the additional cells. In a second step, when the bitstream configuration is written to file, we activate malicious logic by reconfiguring the camouflaging circuitry which actually triggers the Trojan.

The contributions of this paper are the following: 1. To the best of our knowledge, we are the first to publish a concrete attack against FPGA design tools, 2. We introduce the concept of a malicious lookup table (LUT), which is the core element that makes our design flow attack effective and successful, 3. In order to keep our attack undetected by recent detection techniques, we present a camouflaging methodology that preserves malicious functionality without any effect on the legitimate output signals (therefore evading detection techniques relying on unused or rarely used signals or circuits).

## 2. RELATED WORK

Implementing stealthy hardware Trojans has been investigated by several research groups. Sturton et al. present a methodology to design hardware Trojans which evade circuit analysis based on unused circuit identification (UCI) [14]. The approach is effective to evade UCI, however, the circuits which are designed using this methodology are not functionally and formally equivalent to the original circuit. Thus, the triggers designed this way may evade UCI, however, they will not evade exhaustive simulation and formal verification. Also, functional analysis for nearly-unused circuit identification (FANCI) detects these Trojans with high probability [18].

*ICCAD '16, November 07 - 10, 2016, Austin, TX, USA*
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4466-1/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967054

Our approach is different in these aspects: First, we design Trojans that are functionally and formally equivalent to the original hardware description language (HDL) design. As we trigger the Trojan after design-time tests, only bitstream verification or in-circuit testing will reveal our attack. However, this may be extremely hard, as this will require exhaustive testing which is known to require exponential time. Second, no test methodology based on UCI will detect our Trojan, as we camouflage our trigger such that it constantly contributes to the compromised payload signal. As our Trojans are always-on in terms of Trojan activity, neither FANCI detects our attack.

In [23], Zhang, Yuan, and Xu propose to use sequences of patterns in order to evade UCI. However, triggers based on sequences will leave at least one signal rarely used, i.e., the signal which actually activates Trojan payload. Also, the design is functionally and formally altered, which will be revealed by exhaustive simulation and formal verification. In our work, we propose a novel methodology to evade UCI by camouflaging the trigger circuit as useful during design-time testing – when the testing phase is finished, we replace portions of the trigger circuit in order to activate Trojan payload.

Compromised design tools such as compilers have a long history when assessing the trust level of a system, as pointed out by Thompson [16]. Likewise, attacking the field-programmable gate array (FPGA) design flow has attracted the interest of research groups since the rise of the hardware Trojan problem. Roy, Koushanfar, and Markov identify the threats to hardware designs emerging from design tools, such as synthesis tools whose binaries are replaced, or pre-/postprocessing scripts that are altered at a particular site [12].

Chakraborty et al. show that it is possible to attack an FPGA by directly modifying its unencrypted configuration bitstream [2]. The authors classify Trojans that overlap with original functionality as type-II Trojans, in contrast to type-I Trojans which do not overlap and therefore simply can be dropped into unallocated resources. Following this terminology, we focus on malicious functionality which is inserted into existing designs (type-II Trojans). Therefore, our work can be seen supplementary to [2], who only consider type-I Trojans. Another aspect with regard to [2] is that the authors have to attack the placed and routed design, which implies that reverse-engineering is needed in order to include type-II Trojans. We, on the other hand, attack the design at the register transfer level (RTL), which makes our attack way more generic and effective as we do not have to consider physical constraints or the target technology.

In order to prevent the bitstream from tampering, it can be encrypted. Bitstream encryption itself will not be an option to defend against our attack, because the bitstream will already be equipped with malicious functionality when it is encrypted. Protection schemes such as dummy logic insertion as proposed by Khaleghi et al. [7] will fill unused resources in an FPGA and therefore be an effective countermeasure against type-I Trojans presented in [2]. However the method will not prevent our Trojan from being inserted, because when dummy logic is generated, malicious functionality is already part of the design.

Chakraborty et al. [2] argue the strength of their attack with the fact that there are "no verification mechanisms for the correctness of FPGA configuration bitstreams" [2], which is why "such a modification would be extremely difficult to detect pre-deployment" [2]. We also believe that bitstream verification is the key to detect attacks against the FPGA design flow. Backed by Trimberger [17], we propose equivalence checking to reveal our attack. However, many bitstream formats are proprietary. Thus, if the tool chain is untrusted, a vendor-specific bitstream-to-netlist converter is also not trustworthy. Therefore, we call for open bitstream formats that allow third-party verification tools to prove formal equivalence of an FPGA's specification and implementation.

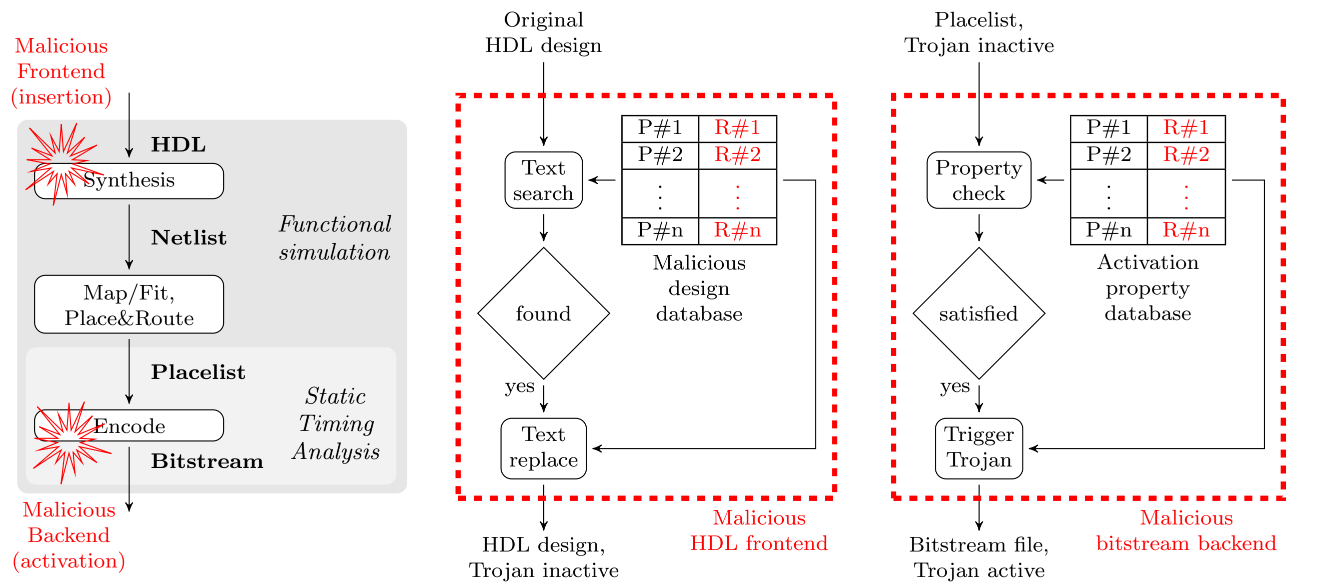
## 3. METHODOLOGY

### 3.1 Threat model

An attacker manages to reverse-engineer a commercial design tool by e.g. disassembling a binary executable and locating targets to inject code [10]. Vendors of electronic design automation (EDA) tools may protect their binaries against reverse-engineering by encryption and obfuscation techniques, which will considerably increase the attacker's effort. However, we consider the effort comparable to reverse-engineering graphic database system (GDSII) layouts, which is a well-accepted threat model in the community. The benefit of compromising EDA tools is higher impact, as a potentially higher number of designs can be compromised. It is also reasonable that an attacker includes malicious code into an open-source tool and compile it to a binary version. Next, a design house is intruded over the Internet and the golden image that serves as installation template for corporate computers is located. Within the golden image, the legitimate binary of the design tool is replaced by its compromised version. This way, multiple machines in the design house are compromised at once. It is also reasonable that an insider replaces the legitimate with the compromised binary. However, for an advanced attacker intrusion over the Internet might be the better option in order to hide the attack and in order to launch multiple attacks against numerous design houses. Skilled attackers and nation state hackers are likely to successfully conduct such an attack [6]. In this context, it is also possible that the design tool vendor collaborates with major intellectual property (IP) vendors in order to include malicious behavior into IP cores during synthesis. When the compromised synthesis tool reads the design, it injects HDL code based on pattern matching and replacing. A novel trigger mechanism is included in order to evade detection during design time. The trigger is fired when the design is written into a bitstream configuration file. As proposed earlier [8, 5, 14], we also believe that an attacker will use a hardware Trojan attack in order to facilitate higher-level attacks. Therefore, the attacker most likely will strive to compromise general-purpose hardware, such as central processing units (CPUs), the memory system, or the storage system [1], rather than application-specific hardware.

### 3.2 The design flow attack in general

Our attack against the FPGA design flow is shaped such that a design formulated in a HDL is slightly modified when read by the synthesis tool. This slight modification does not show up in any output throughout the design flow until the bitstream is written to file. All other output products (such as post-place simulation netlists) are formally equivalent to the original HDL design. This is done with a two-phase



(a) (b) (c)

Figure 1: Function principle of our attack (a) (design flow inspired by [4]). A malicious frontend injects malicious HDL code into an original design based on pattern matching (b). A malicious backend activates the attack by checking properties of specific trigger cells which are reconfigured when properties are satisfied (c)

approach, where malicious functionality is injected in the design flow's HDL frontend and actually activated in the bitstream backend. Figure 1a illustrates a simplified version of our design flow, highlighting the points in the flow where the design is compromised. In our implementation, the injection is carried out by searching and replacing text. Although injected, malicious functionality is still inactive at this stage. Searching and replacing text is useful due to several reasons. First, it is easily implemented using standard routines. Second, it can be applied to a multitude of standard soft third-party intellectual property cores (3PIPs) in order to conduct "standard" hardware Trojan attacks and therefore widely distribute the same Trojan in a variety of deployed systems that incorporate this 3PIP core. Third, high flexibility is guaranteed compared to algorithmic approaches. In our attack model, the possibilities to inject malicious functionality are manifold for the compromised frontend. For instance, entire modules can be matched in order to replace identified functional blocks by a malicious version. Combined with reasoning about the target application of the design, even tailored attacks can be carried out. Figure 1b illustrates the function principle of the malicious frontend. The design is matched against a malicious design database in order to inject malicious functionality. Consider that the database can be implemented such that it is accessible via the Internet over a covert channel in order to perform cutting-edge attacks and to keep the database up to date, once the design house is compromised.

After the HDL frontend further transformed the compromised HDL design, several steps of synthesis and optimization are carried out (that what is actually done by a hardware synthesis tool). Every of these steps alters the design representation and produces output which can be simulated and formally verified. Our attack remains stealthy during the entire design flow, because the compromised design is functionally and formally equivalent to the original HDL representation of the design. At the very last stage of the FPGA design flow, when writing the bit stream configuration to file, the Trojan is "activated". This is done by actually matching properties of specific trigger cells that have been injected by the malicious HDL frontend. These can be, for instance, properties of a cell that are syntactically correct, but semantically impractical. An example of such a property is that all input signals of a cell are connected to each other. Figure 1c illustrates the second stage of our attack. Here, the compromised design is present as a placelist. The malicious backend screens the placelist by checking each cell to satisfy a set of properties that mark a cell malicious. If a cell satisfies such a set of properties, it is reconfigured in order to activate malicious behavior. Replacement rules are necessary for different types of malicious trigger cells. Therefore, for each activation property set, a rule is associated which defines how malicious functionality is activated in the resulting bitstream file.

Summing up, our attack implements malicious functionality with a trigger that behaves fundamentally different than traditional triggers. Our trigger mechanism works such that before the trigger fires, malicious functionality is always on, without propagating compromised signals. This makes it hard, if not impossible, for state-of-the-art detection tools that rely on identifying rarely used trigger signals to detect our attack. After the trigger fired, malicious functionality again is always on, but now compromised signals propagate, therefore violating the specification. Hence, exhaustive functional testing or formal verification of the bitstream file will reveal our attack. However, as pointed out by Chakraborty et al. [2] bitstream verification is not common in today's design flows. Also, common bitstream formats are proprietary and not publicly documented, which makes it hard for third-party verification tool vendors to offer verification solutions. Making public the bitstream formats will heavily increase trust in FPGA systems, as also will do open-source verification tools. However, with public bitstream formats, IP protection and counterfeit issues are to be solved.

### 3.3 Phase 1: Replace functional blocks by malicious versions

In order to demonstrate our attack, we introduce the concept of a *malicious unary operation*. A malicious unary operation is a unary operation that behaves as expected when simulated, but different when actually implemented in hardware such that the original unary operation is replaced by any other unary operation. There are four unary operations with regard to the output: constant zero ("ZRO"), invert input ("NOT"), same value as input ("BUF"), and constant one ("ONE"). We realize the malicious unary operation with a lookup table (LUT). A LUT in general is a memory with a width of one bit, having $n$ inputs that address one of the resulting $2^n$ memory bits. Thus, any logic function can be implemented by configuring the memory bits as specified by the logic function's truth table. The characteristic property of our special LUT is that once the Trojan is triggered in the second phase of our attack, the function of this LUT changes such that it behaves like a another unary operatoin. In this context, we call such a type of LUT *malicious LUT*. The clue about the malicious LUT is that its least significant bit (LSB) and its most significant bit (MSB) serve as effective output of the malicious unary operation, because a further property of the malicious LUT is that its four inputs are connected to each other. Thus, if the input of the original unary operation is 0, all four inputs of the LUT are 0, therefore addressing the LUT's LSB. Likewise, if the input of the original unary operation is 1, the LUT's MSB is addressed. The remaining 14 bits serve as switch in order to define the resulting function of the LUT. We divide the 14 bits into two 7-bit wide words whose values are defined by initialization. Each of these two *configuration words* can take two predefined bit patterns. These patterns are freely chosen. We call the patterns *KEEP* and *FLIP*. We define *KEEP* as $7'b1110001$ and *FLIP* as $7'b1101010$. If the higher-order configuration word (HOCW) is *KEEP*, this means, that in the second phase of our attack, the MSB keeps its value. If the word holds the *FLIP* pattern, this means that the content of the MSB is flipped such that a 0 becomes 1 and vice versa. The same is true for the lower-order configuration word (LOCW), which determines whether the LSB is kept or flipped.

With this setup, we can implement 16 malicious mappings of binary operations. It is almost impossible to find such a LUT in a real-world design, because a synthesis tool will not generate a unary cell, and if it would, then it would only use one cell input. In addition, we assess the probability low that a manually initialized LUT will hold the *KEEP* or *FLIP* patterns on bits 14 to 1. Figure 2 illustrates the malicious unary operation, implemented as a malicious LUT.

Table 1 defines the possible configurations of the malicious LUT. As input $I$ is simultaneously applied to all four inputs of the LUT, we can only address either the LUT's LSB ($I = 0$) or MSB ($I = 1$). If $I = 0$, output $O$ takes the value of the LUT's LSB, whereas $O$ takes the value of the LUT's MSB if $I = 1$. This behavior of the malicious LUT exactly reflects expected behavior of the original unary operation. This fact is indicated by the column $f_{\text{simulation}}$ (for *simulation functionality*). The last column indicates the malicious LUT's function after the second phase of our attack, where the LUT is reconfigured depending on the

Table 1: Possible configurations of the malicious LUT

| MSB | HOCW | LOCW | LSB | Function | |
|---|---|---|---|---|---|
| 15 | 14–8 | 7–1 | 0 | $f_{\text{simulated}}$ | $f_{\text{hardware}}$ |
| 0 | KEEP | KEEP | 0 | ZRO | ZRO |
| 0 | KEEP | FLIP | 0 | ZRO | NOT |
| 0 | FLIP | KEEP | 0 | ZRO | BUF |
| 0 | FLIP | FLIP | 0 | ZRO | ONE |
| 0 | KEEP | FLIP | 1 | NOT | ZRO |
| 0 | KEEP | KEEP | 1 | NOT | NOT |
| 0 | FLIP | FLIP | 1 | NOT | BUF |
| 0 | FLIP | KEEP | 1 | NOT | ONE |
| 1 | FLIP | KEEP | 0 | BUF | ZRO |
| 1 | FLIP | FLIP | 0 | BUF | NOT |
| 1 | KEEP | KEEP | 0 | BUF | BUF |
| 1 | KEEP | FLIP | 0 | BUF | ONE |
| 1 | FLIP | FLIP | 1 | ONE | ZRO |
| 1 | FLIP | KEEP | 1 | ONE | NOT |
| 1 | KEEP | FLIP | 1 | ONE | BUF |
| 1 | FLIP | KEEP | 1 | ONE | ONE |

values in the configuration words (bits 14–8 and 7–1). How this is carried out is treated in Section 3.4.

The trick is that the malicious LUT can be used to derive a trigger signal in order to activate malicious payload injected by the malicious frontend. If doing so, the attacker has to know the target technology in order to inject the correct type of LUT. However, as the attacker compromises a vendor-specific design-flow, she knows which types of LUTs are used in that flow. For our demonstration flow (see Section 4), target-technology cells are treated as black-box cells, which are preserved during optimization passes.

### 3.4 Phase 2: Activate malicious replacement

In this phase of the attack, the malicious LUT is reconfigured such that malicious functionality is activated. This is done by the malicious backend, which writes the synthesized bitstream to a file. The malicious backend works as follows: When the bitstream configuration file is created from the placed and routed design, every cell is checked if it is a malicious LUT by checking the following properties: 1. The LUT's inputs are connected to each other, 2. Bits fourteen to eight of the LUT's initialization is either the KEEP or FLIP configuration pattern, 3. Bits seven to one of the LUT's initialization is either the KEEP or FLIP configuration pattern.

If a cell satisfies all of the above properties, it is consid-

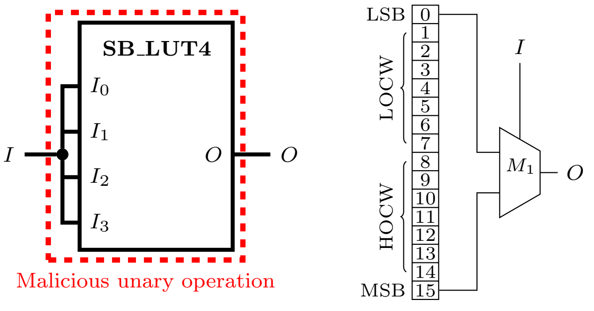


Figure 2: A malicious LUT cell. The four inputs are connected and either the LSB or MSB is addressed (a). LOCW and HOCW determine the value of LSB and MSB after phase two of our attack and hold either the KEEP or FLIP pattern, input $I$ determines the value of output $O$ (b).

ered a malicious LUT. Therefore, the LUT's initialization is changed depending on the malicious LUT's configuration, which is specified by the configuration patterns found in the LOCW and HOCW (see Table 1). This way, the malicious LUT's 16 functions are realized as specified in Table 1. Flipping a bit is a reversible process. Therefore, in order to keep the attack undetected, we flip the flipped bits again after the configuration has been written to file. This way, the in-memory representation of the design in the place-and-route tool employs the untriggered Trojan again. Likewise, a file that reads a bitstream and generates another representation of the design (such as a simulation netlist) could also flip the bits again and restore original behavior.

## 4. DEMONSTRATION

We implemented the design flow attack using the free and open-source iCE40 design flow, which consists of the hardware synthesis suite *yosys* [21, 20], the place-and-route tool *arachne-pnr* [13] and the tools from *Project IceStorm* [19]. We modified the Verilog frontend of yosys in order to perform malicious text replacements[1]. The example designs are processed by yosys using the simple synthesis script given in Listing 1. The synthesis script calls the Verilog frontend in order to read the Verilog design (Line 1) and synthesizes it for the Lattice Semiconductor iCE40 FPGA family (Line 2). In our attack, a malicious frontend injects additional functionality when reading the design. In order to emphasize that the legitimate version of `read_verilog` has been replaced by a compromised version we print the affected lines in Listing 1 and Listing 2 in bold red. Listing 2 shows the tool chain to place and route the design, and to encode it to the binary bitstream format which is used to configure the FPGA. As outlined, a malicious backend triggers the injected functionality, which in our case is the bitstream configuration backend of the place-and-route tool in our tool chain. We modified the method `write_txt()` of *arachne-pnr*, such that it looks for cells that satisfy a set of Trojan activation properties and then replaces the untriggered version of the cell by the triggered version.

Listing 1: Yosys synthesis script applied to the example designs

```
1 read_verilog instruction-decoder.v
2 synth_ice40 -blif instruction-decoder.blif
```

Listing 2: Commands for place-and-route and encoding

```
arachne-pnr -o instruction-decoder.asc instruction-
    decoder.blif
icepack instruction-decoder.asc instruction-decoder.
    bin
```

We demonstrate our attack with a simplified instruction decoder of a standard CPU which is to be compromised by a privilege escalation attack. For the sake of simplicity, we only show the relevant part of the decoder, i.e. the privilege checker that determines if an instruction requires superuser privileges. We state that in a CPU, there are privileged instructions and unprivileged instructions, and that there is a facility in the instruction decoder, that determines if a certain instruction, defined by it's opcode, requires root privileges in order to be executed. The Verilog code to describe the original version of this privilege checker is given in Listing 3, the corresponding schematic is given in Figure 3, comprising cells $C_1$, $C_2$, and $G_1$. The function principle of the instruction decoder is simple: Based on the opcode, the decoder checks whether the instruction requires superuser privileges. If yes, it sets the flag *requires_superuser* true, and false otherwise. In our CPU, an instruction requires superuser privileges if the most significant bytes of the opcode are `0xAB`, or `0xCDEF`.

Listing 3: Verilog model of the original instruction decoder (`instruction-decoder.v`)

```
module instruction_decoder (
    input [31:0] opcode,
    output reg requires_superuser
);

    always @* begin
        casez (opcode)
            32'hABzzzzzz, 32'hCDEFzzzz:
                requires_superuser = 1;
            default:
                requires_superuser = 0;
        endcase
    end
endmodule
```

In our example attack, we include an instruction, which normally requires superuser privileges, but for which we explicitly turn off this requirement.

In order to effectively mount the attack in a deployed Trojan, the attacker will need a piece of software on the compromised machine that executes this hidden instruction, which for instance could be an instruction that swaps the page tables. This way, it would become possible for the process to escape a hypervisor, which could potentially harm the integrity and confidentiality of entire datacenters.

As the attacker does not like to be detected, malicious behavior is masked such that during simulation and/or formal verification, every instruction behaves as intended. This is implemented using a simple `if` statement (Line 22 in Listing 4). Note that the designer does not recognize the additional Verilog code, as it is injected in the Verilog fronted of the synthesis tool. In the design flow we use for our experiments, injection happens in Line 1 in Listing 1. Thus, malicious functionality is only present in the synthesis tool's internal representation of the design. Listing 4 shows the Verilog model of the compromised instruction decoder. Malicious parts are written in bold red. So, what happened here? In Lines 10 to 18, a malicious LUT is instantiated and initialized using the configuration patterns defined in Lines 5 to 6. Table 1 reveals that the malicious LUT is configured such that in simulation, it implements a constant zero, whereas in hardware, it implements a constant one. All four inputs of the LUT are connected to the opcode's LSB, which more or less is an arbitrary choice, as the LUT's output in it's current configuration does not depend on the LUT's inputs. The output of the malicious LUT is used as a trigger signal in order to activate the Trojan payload. The payload of our Trojan is implemented in Lines 22 to 32. If the trigger signal, `trojan_active`, is high, privilege checking is manipulated for one specific instruction, which means that one privileged instruction can be executed without having superuser privileges. This instruction has opcode `0xABCDEF01` (Line 25). If the trigger signal is low, the instruction decoder works as

[1]The source code for the modified versions of the design tools as well as example designs are available from the second author's web site: http://www.clifford.at/papers/2016/malicious-lut/

specified (Lines 32 to 39) in Listing 3. The clue is that the trigger signal is always low throughout the design phase, thus the compromised instruction decoder works as specified and is functionally equivalent to the original Verilog code. Only when the design is written to a bitstream configuration file, the trigger signal becomes high, which permanently triggers the Trojan payload.

Listing 4: Verilog model of the compromised instruction decoder. Malicious functionality written in bold red is injected by the compromised Verilog frontend of the design flow (`read_verilog`)

```
 1 module instruction_decoder (
 2     input [31:0] opcode,
 3     output reg requires_superuser
 4 );
 5 localparam [6:0] KEEP = 7'b1110001;
 6 localparam [6:0] FLIP = 7'b1101010;
 7
 8 wire trojan_active;
 9
10 SB_LUT4 #(
11     .LUT_INIT({1'b0, FLIP, FLIP, 1'b0})
12 ) magic_zro_one (
13     .I0(opcode[0]),
14     .I1(opcode[0]),
15     .I2(opcode[0]),
16     .I3(opcode[0]),
17     .O(trojan_active)
18 );
19
20 always @* begin
21     // A MUX switching between HT and original code
22     if (trojan_active) begin
23        // This is the code with the HT enabled
24          casez (opcode)
25             32'hABCDEF01:
26                 requires_superuser = 0;
27             32'hABzzzzzz, 32'hCDEFzzzz:
28                 requires_superuser = 1;
29             default:
30                 requires_superuser = 0;
31         endcase
32     end else begin
33         // This is the original code
34         casez (opcode)
35             32'hABzzzzzz, 32'hCDEFzzzz:
36                 requires_superuser = 1;
37             default:
38                 requires_superuser = 0;
39         endcase
40     end
41 end
42 endmodule
```

The backend first checks every cell if it implements a malicious LUT. A cell qualifies as malicious LUT if it satisfies the properties specified in Section 3.4. The malicious LUT specified in Listing 4, Lines 10 to 18, satisfies all the properties, therefore it's functionality is changed according to it's configuration words. The LUT's MSB and LSB both are 0. As both configuration words indicate FLIP, both the MSB and the LSB are flipped, such that afterwards their value is 1. This means, that regardless of the input, the ouput is constant 1. Therefore, the trigger signal `trojan_active` becomes true and from this moment, the Trojan payload functionality is always on as can be seen from Listing 4, Lines 22 to 32.

## 5. DISCUSSION

We presented a hardware Trojan attack, where malicious functionality is inserted by a compromised frontend which is stealthy throughout the entire design flow, until written to

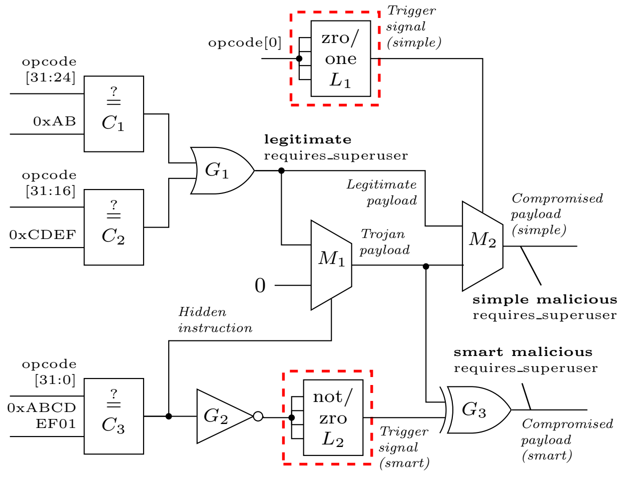


Figure 3: Schematic of the demonstration example

a bitstream file. In our attack, Trojan payload is triggered by a compromised backend which reconfigures malicious LUTs. While exhaustive simulation will not reveal the attack, certain approaches to hardware Trojan detection have been presented that might be able to do so. In the following, we qualitatively evaluate our attack regarding its detectability and present obfuscation techniques in order to evade state-of-the-art design-level hardware Trojan detection.

### 5.1 Detectability of our trigger

State-of-the-art techniques in design-level hardware Trojan detection leverage the fact that Trojan payload is triggered by rare events. Thus, if a signal is rarely activated, it can serve as a trigger and therefore can be considered malicious.

Hicks et al. propose to use unused circuit identification (UCI) in order to identify such signals by analyzing the data flow graph (DFG) of a design [5]. Data flow is defined such that a signal can have impact on another signal, i.e., the input signal of a gate can have impact on its output signal. Functional simluation is used in order to reason on the effective impact. If a dependent signal does not change on variations of a driving signal throughout the simulation, the circuit in between is considered *unused*, therefore potentially malicious.

In order to identify *nearly* unused circuits, Waksman, Suozzo, and Sethumadhavan propose to apply functional analysis for nearly-unused circuit identification (FANCI) to electronic designs at design level. Instead of the DFG, the truth tables of all gates are analyzed in order to find inputs that very rarely impact output signals. This is achieved by computing a *control value*, which is a metric for an output signal's activity as a function of any of its input signals. A signal is considered malicious when the control value remains under a user-defined threshold.

Rather than unused circuits, Zhang et al. propose VeriTrust in order to identify unused *inputs* in an electronic design [22]. VeriTrust is a dynamic approach performed on top of functional simulation. A tracer records activation behavior and flags inputs that are potentially redundant. As functional simulation is not exhaustive, a checker determines if such a signal indeed is redundant in order to flag it suspicious.

Figure 3 shows the schematic of our demonstration example. Original functionality is implemented by cells $C_1$, $C_2$ and $G_1$, whereas malicious functionality is implemented by cells $C_3$, $M_1$, $M_2$, and $L_1$. The output signal of malicious lookup table (LUT) $L_1$ serves as the trigger signal, which drives multiplexer $M_2$ in order to switch between legitimate and Trojan payload (such as in [5], Fig. 7). When untriggered, $M_2$'s control input is constant zero, which means that only the first input of $M_2$ affects its output and that its second input never is switched as the output signal. Therefore, $M_2$ becomes unused and will be flagged suspicious by UCI, FANCI and VeriTrust.

FANCI will also flag comparator $C_3$ malicious, as its output will only be 1 if the opcode of our example central processing unit (CPU) is `0xABCDEF01`, which will happen by a chance of $1/2^{32} \approx 0.23 \cdot 10^{-12}$. However, using the malicious LUT as a trigger that masks malicious behavior, we could also use comparators $C_1$ or $C_2$ in order to exhibit malicious behavior, which would result in $2^{24}$ possible instructions with modified privileges for comparator $C_1$ (with control value $2^{24}/2^{32} \approx 3.91 \cdot 10^{-3}$), and for $C_2$, $2^{16}$ instructions (with control value $2^{16}/2^{32} \approx 15.26 \cdot 10^{-6}$). As also pointed out by the authors of FANCI, a less well-hidden trigger is more likely to evade detection by FANCI, but is also more likely to be detected by standard validation testing [18]. However, until triggered, our Trojan is functionally and formally equivalent to the original hardware description language (HDL), therefore evading detection by standard validation testing.

With the malicious LUT, we are able to implement triggers which are by far more potent as the simple version presented above. In the next section, we present an improved version of our trigger mechanism that renders our attack stealthy.

## 5.2 An improved trigger mechanism

In order to evade UCI, FANCI and VeriTrust, we mask our trigger scheme such that at any time, every input signal contributes to the output signal to the same extent. In the lower portion of Figure 3, we present an improved version of the Trojan we introduced previously. The Verilog code of the improved Trojan is given in Listing 5. We take a different configuration for malicious LUT $L_2$ such that it behaves like an inverter during simulation and as constant zero in hardware ("not/zro"). We negate the condition that exhibits Trojan behavior (i.e., "the hidden instruction is executed") which is checked by comparator $C_3$ and feed it to the malicious LUT's input. During simulation, the LUT again negates the signal ("not"). Thus, if the hidden instruction is executed, this is reflected as a 1 at the malicious LUT's output. Likewise, multiplexer $M_1$ outputs 0 if the hidden instruction is executed, as its control input is driven by comparator $C_3$'s output. However, if this is the case, $M_1$'s behavior would violate the specification (when directly used as an output), which can be detected by functional simulation. Therefore, we mask $M_1$'s output by xor'ing (cell $G_3$) it with malicious LUT $L_2$'s output, which is 1 under this condition. As a result, $G_3$'s output is 1, which reflects specified behavior ($0 \oplus 1 = 1$). Neither exhaustive functional nor formal verification will detect the attack at this stage, because the Trojan does not yet act maliciously. Also, UCI, FANCI and VeriTrust will not detect the trigger, as all involved gates steadily contribute to the compromised payload.

Listing 5: Verilog model of an improved trigger mechanism that cannot be detected by state-of-the-art design-level hardware Trojan detection techniques such as UCI, FANCI, or VeriTrust

```verilog
module instruction_decoder (
  input [31:0] opcode,
  output reg requires_superuser
);
  localparam [6:0] KEEP = 7'b1110001;
  localparam [6:0] FLIP = 7'b1101010;

  wire opcode_is_not_ABCDEF01 = opcode != 32'
     hABCDEF01;
  wire opcode_is_ABCDEF01_and_zero_in_hw;

  SB_LUT4 #(
    .LUT_INIT({1'b0, KEEP, FLIP, 1'b1})
  ) magic_not_zro (
    .I0(opcode_is_not_ABCDEF01),
    .I1(opcode_is_not_ABCDEF01),
    .I2(opcode_is_not_ABCDEF01),
    .I3(opcode_is_not_ABCDEF01),
    .O(opcode_is_ABCDEF01_and_zero_in_hw)
  );

  always @* begin
    casez (opcode)
      32'hABCDEF01:
        requires_superuser = 0;
      32'hABzzzzzz, 32'hCDEFzzzz:
        requires_superuser = 1;
      default:
        requires_superuser = 0;
    endcase

    requires_superuser = requires_superuser ^
     opcode_is_ABCDEF01_and_zero_in_hw;
  end
endmodule
```

Once the placelist is written to file as a configuration bitstream, the Trojan is triggered by the malicious backend. Malicious LUT $L_2$ is reconfigured such that it no longer behaves as inverter, but as a driver for constant zero. Thus, the output of XOR gate $G_3$ will be the same as the output of multiplexer $M_1$, enabling malicious behavior to propagate to the module's output (because $A \oplus 0 = A$). Table 2 shows the truth table of both the simple and smart versions of our trigger. We see that for the simple version, when not triggered, the trigger signal is always 0 and therefore does not contribute to the output signal *compromised payload*. For the smart version, on the other hand, the trigger signal contributes to the compromised payload signal, which therefore cannot be identified as a trigger signal by FANCI [18]. Also UCI [5] will not identify our smart trigger as potentially malicious, as the trigger signal affects the compromised output. Neither does VeriTrust [22], as all inputs are used and therefore no input will be flagged redundant.

Post-implementation simulation and on-chip testing is the only way to reveal our trigger. However, this would require exhaustive testing, which is known to be practically unfeasible. Therefore, it is very unlikely that our trigger is revealed this way, which is $2^{-32}$ in the given example.

## 5.3 Preliminaries for the verification tool

As we have seen, it is possible for verification tools to detect simple un-obfuscated triggers proposed in this work. However, such a tool must follow some guidelines in order to be effective: First, the verification framework must use the same HDL frontend as the design flow. Second, if the verification method is carried out on top of functional simulation (such as [5, 22]), test vectors must be chosen

Table 2: Truth table of our attack (simple and smart triggers). The signal names in the table correspond to the signal names written italic in Figure 3

| | | | **Simple** | | | | **Smart** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Triggered | | | No | | Yes | | No | | Yes | |
| Hidden instruction | Legitimate payload | Trojan payload | Trigger signal | Compromised payload | Trigger signal | Compromised payload | Trigger signal | Compromised payload | Trigger signal | Compromised payload |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

wisely in order to exhibit malicious behavior of the design. Third, in order to avoid a large number of false positives, the point in the flow must be chosen such that verification is carried out before certain optimization is carried out. E.g., resource sharing creates weakly affecting or unused inputs which are flagged suspicious, although being legitimate.

## 6. CONCLUSIONS AND FUTURE WORK

Formal methods are a great tool in order to prove that the implementation of an electronic design behaves as specified [11, 9]. Hence, we identify equivalence checking as adequate measure in order to reveal manipulations of the bitstream configuration, which is well-known state of the art. However, the big problem is that bitstream verification is not a popular task in today's field-programmable gate array (FPGA) design and verification flows. One of the major reasons for this fact is that bitstream formats are not publicly documented, which makes it hard for third-party verification tool vendors to offer solutions which prove that the bitstream configuration is formally equivalent to the original HDL description. We have shown that it is easily possible to inject malicious behavior into electronic designs using compromised design tools without being noticed by neither the designer nor state-of-the-art tools targeted at Trojan detection. Therefore, we call for open and publicly documented bitstream formats in order to increase trust in FPGA systems. Provided that bitstream formats are publicly documented, it is easy for an equivalence check between the bitstream configuration and the placelist to reveal the attack we presented in this paper. However, once the design complexity increases of both the design and malicious functionality, more sophisticated approaches to equivalence checking are needed. First, equivalence between the bitstream configuration and the placelist must be ensured. In a second step, equivalence between the placelist and the original HDL description can prove that the design can be trusted. We will address these issues in the future in order to demonstrate the feasibility of effective FPGA bitstream verification.

## Acknowledgments

This paper is supported by TU Wien research funds.

## References

[1] M. S. Anderson, C. J. G. North, and K. K. Yiu. *Towards Countering the Rise of the Silicon Trojan*. Tech. rep. Dec. 2008.

[2] R. S. Chakraborty et al. "Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream". In: *IEEE Design Test* 30.2 (2013), pp. 45–54.

[3] R. Chakraborty, S. Narasimhan, and S. Bhunia. "Hardware Trojan: Threats and emerging solutions". In: *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International.* 2009, pp. 166–171.

[4] S. Drimer. *Security for volatile FPGAs*. Tech. rep. University of Cambridge, 2009.

[5] 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.

[6] R. Joyce. *Talk: Disrupting Nation State Hackers*. Talk given at USENIX Enigma Security Conference. Jan. 27, 2016.

[7] B. Khaleghi et al. "FPGA-Based Protection Scheme against Hardware Trojan Horse Insertion Using Dummy Logic". In: *IEEE Embedded Systems Letters* 7.2 (2015), pp. 46–50.

[8] S. T. King et al. "Designing and implementing malicious hardware". In: *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. San Francisco, California: USENIX Association, 2008, pp. 1–8.

[9] C. Krieg, M. Rathmair, and F. Schupfer. "A Process for the Detection of Design-Level Hardware Trojans Using Verification Methods". In: *Proceedings of the 11th IEEE International Conference on Embedded Software and Systems (ICESS 2014)*. Aug. 2014, pp. 741–746.

[10] C. Peikari and A. Chuvakin. *Security Warrior*. Ed. by M. Loukides. O'Reilly Media, Inc., 2004.

[11] M. Rathmair, F. Schupfer, and C. Krieg. "Applied formal methods for hardware Trojan detection". In: *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on.* 2014, pp. 169–172.

[12] J. Roy, F Koushanfar, and I. Markov. "Extended abstract: Circuit CAD tools as a security threat". In: *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on.* 2008, pp. 65–66.

[13] C. Seed. *Arachne-pnr*. URL: https://github.com/cseed/arachne-pnr.

[14] C. Sturton et al. "Defeating UCI: Building Stealthy and Malicious Hardware". In: *Proc. IEEE Symp. Security and Privacy (SP)*. 2011, pp. 64–77.

[15] M Tehranipoor and F Koushanfar. "A Survey of Hardware Trojan Taxonomy and Detection". In: *Design Test of Computers, IEEE* 27.1 (2010), pp. 10–25.

[16] K. Thompson. "Reflections on Trusting Trust". In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763.

[17] S. Trimberger. "Trusted Design in FPGAs". In: *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE.* 2007, pp. 5–8.

[18] 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.

[19] C. Wolf. *Project IceStorm*. URL: http://www.clifford.at/icestorm/.

[20] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/. URL: http://www.clifford.at/yosys/ (visited on 03/10/2016).

[21] C. Wolf and J. Glaser. "Yosys - A Free Verilog Synthesis Suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 2013.

[22] J. Zhang et al. "VeriTrust: Verification for Hardware Trust". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 34.7 (2015), pp. 1148–1161.

[23] J. Zhang, F. Yuan, and Q. Xu. "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 153–166.