Reflections on Trusting TrustHUB

Christian Krieg
Institute of Computer Technology (ICT), TU Wien
Gusshausstrasse 27–29/384, 1040 Vienna, Austria
E-Mail: christian.krieg@alumni.tuwien.ac.at

***Abstract*—Hardware security verification is a critical task in evaluating algorithms and tools for effectiveness and efficiency, both from an organizational and a technical view. In the past decade, the TrustHUB benchmark suite evolved as the de-facto standard benchmark suite in order to perform experiments that allow quantitative analysis of countermeasures, but also to provide statistical data on effectiveness, and performance indicators to assess practical applicability. In this work, we study the effectiveness of the TrustHUB benchmark suite, and its ability to provide meaningful and reasonable experimental data on algorithms that target security properties of digital designs. We systematically elaborate fundamental properties of effective hardware Trojan benchmarks: Correctness, Maliciousness, Stealthiness, and Persistence. We utilize these properties to classify a reasonable subset of the TrustHUB benchmark suite by means of effectiveness. The results of our study are fairly surprising: Only three out of 83 studied benchmark designs can be considered actual hardware Trojans, all others entail undocumented limitations that may impact the results of experiments that evaluate countermeasures. We introduce a comprehensive, yet simple yes/no effectiveness classification framework, with which we identify potential peculiarities that may render most of these benchmarks ineffective. We identify spots of improvement, and provide recommendations for effective benchmark design.**

I. Introduction

A. Motivation

Hardware Trojan research is a delicate field of cyber security research, because it mainly depends on assumptions. While, e.g., for network-based attacks real data is available in order to reason about currently existing threats, this is not the case for hardware Trojan attacks. Because no real incidents have been reported so far, everything is based on assumptions in hardware Trojan research. This entails two major problems: (1) First, by assuming how potential hardware Trojan attacks may be carried out, we as a research community become part of the problem, because by publishing potential attacks, we show others (i.e., potential attackers) how to do it. (2) Second, and more severe, when relying on assumptions only, it is important that these assumptions are verifiable to hold in realistic settings. If this can not be assured, the effectiveness of countermeasures is questionable that have been developed against these assumptions.

The past decade has shown that software supply chain attacks became reality, and numerous hardware security vulnerabilities and attacks have been published [1, 14, 16, 5]. It is therefore more important than ever to establish reliable methods for hardware security verification. The effectiveness of such methods is usually tested against benchmark designs; In the field of hardware Trojans, the TrustHUB benchmark suite serves as the de-facto standard for evaluating the effectiveness of hardware Trojan detection methods. In the following, we identify several limitations of the TrustHUB benchmarks that may appear counter-intuitive to a hardware security researcher. These limitations potentially limit the expressiveness of evaluation results, which may result in misleading conclusions. The TrustHUB hardware Trojan benchmark suite has been existing for over a decade, is widely used, and is the only collection of systematically organized benchmarks which covers a wide range of designs and attacks. Because these benchmarks are so important to the hardware security community, our intention is to provide a basis for making informed decisions when selecting benchmarks for experimental evaluation. Also, we would like to stress the importance of the TrustHUB benchmark suite as a unique educational resource for hardware Trojan design and implementation. We therefore would like to contribute to this valuable resource by providing pointers to build upon these benchmarks in order to extend their scope to experiments that may rely on different assumptions and test methods than intended by the original benchmark designers.

B. Problem statement

While performing experiments, we noticed irregularities and unexpected behavior for some of the TrustHUB benchmarks. After analyzing the benchmarks in more detail, we reason that it may not be appropriate to use the TrustHUB benchmarks without carefully examining their suitability for a specific experiment. A majority of the TrustHUB benchmarks relies on assumptions that we consider unlikely to hold in realistic settings, and which may appear counter-intuitive to potential benchmark users. For instance, many malicious parts are not synthesizable, which – strictly speaking – means, that these benchmarks are no hardware Trojans by definition. The TrustHUB benchmarks have been used for over a decade now, and yet their limitations have not been publicly documented.

C. Contributions

This work studies the effectiveness of a reasonable subset of the TrustHUB benchmarks, and their ability to evaluate hardware Trojan detection methods. We introduce a comprehensive, yet simple *yes/no* effectiveness classification framework, with which we expose limitations that potentially render most of the studied benchmarks ineffective. Also, we highlight assumptions made by benchmark designers that we consider very unlikely to hold in a real hardware Trojan attack. We analyze security implications of such discrepancies, and provide recommendations for effective hardware Trojan benchmark design. This is a fully reproducible paper. We fully disclose our methods, and provide command lines to generate results; all benchmarks are freely available from the original resources (references are provided).

D. Hardware Trojans

A Trojan system is a system that serves a shadow purpose (i.e., *malicious functionality*) besides its intended (i.e., *benign*) functionality. Trojan systems can be implemented on many abstraction levels, both in hardware and in software. If malicious functionality is implemented in hardware, the system's hardware is called a *hardware Trojan*. Due to resource limitations, we consider a combined software/hardware attack more reasonable than a Trojan attack purely implemented in hardware. For instance, a compromised microprocessor may implement hidden, malicious instructions. Because these instructions are not publicly documented, (benign) software

 Presented at the International Conference on Computer-Aided Design (ICCAD), October 29–November 2, 2023, San Francisco, California, USA. https://doi.org/10.1109/ICCAD57390.2023.10323782

executed by the microprocessor will call these hidden instructions very unlikely. Therefore, also some piece of compromised software may be needed to utilize malicious hardware.

The attacker's ultimate goal is to hide malicious *payload* during design time, to become effective during runtime (post-deployment). Therefore, some kind of *trigger* mechanism is used to avoid detection during functional verification and testing throughout the design and verification flow. To avoid accidental activation, triggers are activated on conditions which are very unlikely to occur during normal operation. Trigger conditions are usually based on rare values or rare sequences that are under the control of the attacker, or reliably and predictably occur at pre-defined "points" in the system's lifetime (e.g., a time bomb [14], or activating malicious inclusions with malicious design tools [8], or leveraging simulation/synthesis mismatch [9]).

Hardware Trojans can either be *functional*, which means they include specific (malicious) functionality (e.g., escalating privileges, or leaking data), or *parametric*, where physical parameters (e.g., timing, power) are modified in order to inject faults, or to perform denial of service (DoS) attacks by reducing the physical health of infested chips.

Depending on the point of inserting malicious functionality in the design and fabrication flow, we distinguish between *design-level* (or, pre-silicon) hardware Trojans, and *physical-level* (or, post-silicon) hardware Trojans. For design-level hardware Trojans, the design files are compromised. For physical-level hardware Trojans, the physical hardware, or the process to fabricate physical hardware, is compromised.

E. TrustHUB benchmarks

We already discussed that no real hardware Trojan attack has been reported so far, so as a consequence, no real data is available for experiments to evaluate the quality and performance of hardware Trojan detection methods. This is why the TrustHUB benchmarks [12, 13] have been introduced, which is a remarkable set of hardware Trojan implementations that give valuable insight into potential trigger and payload mechanisms. The TrustHUB benchmarks feature 17 design classes of variable complexity, including cryptographic accelerators (Advanced Encryption Standard (AES), Rivest-Shamir-Adleman (RSA)), micro-controllers (MC8051, PIC16F84), memory controllers (*memctrl*) communication controllers (EthernetMAC10GE, RS232), bus interconnects (wb_conmax), graphics controllers (vga_lcd), benchmarks of other benchmark suites like *ITC99* (b15, b19), and *ISCAS89* (s15850, s35932, s38417, s38584), and cores with undocumented functionality (MultPyramid), and automatically generated benchmarks (*TRIT*), totalling 106 benchmarks.

Table I provides an overview over the TrustHUB benchmark suite. Each design class provides variants of the original, unmodified (benign) design, with malicious functionality injected. Each design class hosts sets of trigger and payload mechanisms. Combinations of trigger and payload set elements build distinct hardware Trojan attacks, following different threat and attack models. The TrustHUB benchmarks are implemented at various abstraction levels (including Verilog or VHSIC hardware description language (VHDL) designs at register transfer level (RTL), gate-level netlists, technology-mapped netlists (TNL), and layout-level designs (which implement parametric Trojans only). Table I also includes popular examples of selected work utilizing the TrustHUB benchmarks over the past decade, including the static detection method FANCI [15], the dynamic detection method VeriTrust [18], the formal detection framework HaTCh [4], and the machine-learning-based detection approach VIPR [3] .

TABLE I
AN OVERVIEW OVER THE TRUSTHUB BENCHMARK SUITE

Design class	**#**	**Language**	**Level**	**[15]**	**[18]**	**[4]**	**[3]**
AES	27	Verilog	RTL	○	○	○	○
AES	1	Verilog	TNL	○	○	○	○
BasicRSA	4	VHDL	RTL	○	○	●	○
b15	4	Verilog	TNL	○	○	●	○
b19	5	Verilog	TNL	○	○	○	○
EthernetMAC10GE	6	DEF	Layout	○	○	○	○
EthernetMAC10GE	4	Verilog	TNL	○	○	○	○
MC8051	7	VHDL	RTL	○	●	○	○
memctrl	1	Verilog	RTL	○	○	○	○
MultPyramid	2	DEF	Layout	○	○	○	○
PIC16F84	4	Verilog	RTL	○	●	○	○
RS232	14	Verilog	RTL	○	●	●	○
RS232	11	Verilog	TNL	●	○	●	●
s15850	1	Verilog	TNL	●	●	●	○
s35932	3	Verilog	TNL	●	●	●	○
s38417	3	Verilog	TNL	●	●	●	●
s38584	3	Verilog	TNL	○	●	●	○
TRIT	2	Verilog	TNL	○	○	○	○
vga	1	Verilog	TNL	○	○	●	○
wb_conmax	2	Verilog	RTL	○	○	●	○
wb_conmax	1	Verilog	TNL	○	○	●	○
TOTAL	106 **(83 relevant)**						

F. Limitations of this study

This study only considers functional Trojan benchmarks, and does not consider parametric Trojan benchmarks. Parametric Trojan benchmarks are provided without Trojan-free versions to validate any detection efforts. Also, we are not aware of any work that actually uses these benchmarks,[1] which is why we consider them less relevant than the functional Trojan benchmarks. Of the remaining 98 designs, we study a reasonable subset of 83 designs, skipping the *b15*, *b19*, *EthernetMAC10GE*, and *s15850* benchmarks (provided as technology-mapped netlists), because similar malicious functionality is covered by other tech-mapped designs which are included in this study. Also, we do not consider *TRIT* benchmarks, as they are designs whose functionality is not documented, with randomly injected triggers. Their payloads alter random signals. Therefore, our methodology is not applicable, because our threat model assumes a (semi-)human attacker whose intention is to compromise the design's security (i.e., confidentiality, integrity, availability).

II. METHODOLOGY

We first define the threat model we assume for hardware Trojan benchmarks. Also, we define a standard use-case scenario for users of hardware Trojan benchmarks. Based on both the threat model and the use-case scenario, and by taking into account the definition of hardware Trojans, we identify fundamental properties for hardware Trojan benchmarks, which help to model realistic attack scenarios. We use these fundamental Trojan properties to model a (Boolean) *Effectiveness* property, which provides a simple *yes*/*no* answer to the question whether a hardware Trojan benchmarkis effective by design.

[1]We performed an advanced search in the IEEE Xplore® Digital Library with the search term: *"Full Text & Metadata":MultPyramid* for the years between 1884 and 2024. The search yielded three results, while none of them were reporting about experiments with said benchmarks. (We assume that if a paper would report about experiments using *MultPyramid* parametric benchmarks, that these experiments would also include the *Ethernet* parametric benchmarks, as they are the only two parametric Trojan benchmark classes in the TrustHUB benchmark suite.)

A. Threat model

A hardware Trojan benchmark represents a (potentially) compromised intellectual property (IP) core, intended for integration into system-on-chip (SoC) designs. The compromised IP core is designed such that it is a drop-in replacement for a benign version of the IP core, and that no additional manual intervention is needed to integrate the malicious core instead of the original core (e.g., by modifying a top module, or by connecting extra inputs/outputs). The compromised IP core conforms to the specification of the original design (except for malicious inclusions), because a potential user integrates the IP core based on vendor-provided documentation like data sheets and application notes. If the compromised IP core would not conform to the original specification, an IP integrator may immediately detect the attack (which is what the attacker rigorously aims to avoid).

The attacker can be a malicious third-party IP vendor, a malicious insider in a design house, or an outsider who successfully compromises the development life cycle to inject hidden malicious functionality into original design files (thereby designing hardware Trojans). Besides malicious injections, such attacks would take into account revision control systems, automated continuous integration systems and manual code reviews. The attacker is a highly-skilled integrated-circuit design and/or verification engineer with profound knowledge in the cyber-security domain. The attacker designs controllable hardware Trojans, and is able to deterministically activate an attack.

B. Use-case scenario

Hardware Trojan benchmarks are used by hardware security researchers to evaluate the effectiveness and performance of hardware Trojan detection methods they may develop. Besides being used by security researchers, benchmarks are used by (hardware security) verification engineers to characterize detection methods, with the intention to identify those methods which best fit their environment (i.e., threat model, designs, and existing verification flows).

Hardware Trojan benchmark users assume that benchmarks are subject to the threat model defined above (Section II-A), and learn from the benchmark results about the effectiveness and performance of their evaluated Trojan detection tools.

C. Fundamental Trojan properties

We define the following four fundamental Trojan properties, based on the definition of hardware Trojans given in Section I-D, the threat model defined in Section II-A, and the use-case scenario defined in Section II-B. It is imperative that hardware Trojan benchmarks for scientific research satisfy all of these properties. If only one property is violated, the benchmark cannot be used for yielding expressive experimental results when evaluating hardware Trojan detection methods. Please note that these properties are mostly applicable to design-level hardware Trojans.

1) Correctness: A hardware Trojan correctly implements its specification. It is well-tested, and obeys good design practice in order to avoid warnings and/or error messages from design and verification tools (e.g., latch inference). The Trojan behaves as specified; There is a specification for the original design, and also one for the compromised design. In a real setting, the compromised design specification is only known by the attacker, and is not publicly available.

2) Maliciousness: A hardware Trojan acts maliciously, which means that it is specified to compromise the confidentiality, integrity, and/or availability of its host system, and/or of the data processed by the host system. It is technically feasible to deliberately activate malicious functionality **in hardware** as specified (e.g., trigger signals must be accessible by the attacker to trigger the payload).

3) Stealthiness: Malicious functionality stays hidden during the design and verification flow, and the attacker minimizes any chance of potential suspicion. This includes mimicking the coding style of the original design, taking care of appropriate indentation (*tab* vs. *space*) and also to apply the naming scheme that is used in the original design. The attacker uses the minimum amount of code needed to implement malicious functionality. Hardware Trojans comply with the specification of the original design (except, of course, for the malicious part – which is extremely unlikely to manifest during functional verification). Malicious functionality does not introduce resources that impact design structure which is exposed to user interfaces (e.g., additional modules, additional files or directories, additional primary input/output ports, additional *top* modules, etc.). At runtime, it is very unlikely that malicious payload is accidentally activated. Stealthiness is an inherent property of hardware Trojans, as highlighted by the authors of the TrustHUB benchmarks [13, p. 95].

4) Persistence: A hardware Trojan implements malicious functionality **in hardware**, so it must use synthesizable code, and, equally important, survive synthesis and routine circuit optimization passes.

D. Classification framework

We introduce a theoretically motivated practical classification framework to classify the effectiveness of potentially malicious circuits. We identifiy four fundamental properties of hardware Trojan benchmarks: *Correctness* C, *Maliciousness* M, *Stealthiness* S, and *Persistence* P. The fundamental Trojan properties are modeled as Boolean variables, either taking value *True* or *False*. If all fundamental Trojan properties are satisfied (i.e., they all evaluate to *True*), the Trojan is considered *effective by design*, and as a consequence, its *Effectiveness* property E evaluates to *True*. If at least one of the fundamental Trojan properties evaluate to *False*, as a consequence, the *Effectiveness* property also evaluates to *False*. The relation between the *Effectiveness* property and the fundamental Trojan properties can be modeled by a simple Boolean expression, as given in Equation (1).

$$E = C \wedge M \wedge S \wedge P \tag{1}$$

Using this framework, we perform an in-depth study on the effectiveness of a considerable subset of the TrustHUB benchmark suite. We review each Trojan benchmark in detail following a predefined process, and assign *True* or *False* values to Trojan properties according to the results of this process. This way, we transparently and reproducibly identify hardware Trojans being either effective or ineffective by design.

E. Review process

In the following, we briefly outline the process for reviewing each benchmark for this study. A detailed example applies this process in Section III. We basically perform the following steps, while targeting violations of fundamental properties within each step:

1) Review original/compromised specifications
2) Check file structure vs. original specification
3) Check top-level design hierarchy vs. original specification
4) Verify functional behavior:
   a) Compromised design vs. original specification
   b) Compromised design vs. compromised specification
5) Review source code and check for anomalies
6) Summarize results, assign *Effectiveness* property

```
read_verilog aes_128.v
read_verilog lfsr.v
read_verilog round.v
read_verilog table.v
read_verilog top.v
read_verilog Trojan_Trigger.v
read_verilog TSC.v
prep -auto-top
opt -full
show A:top
```

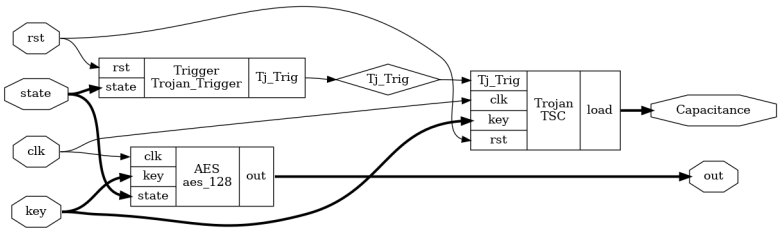

Fig. 1. A top-level block diagram of the compromised design, generated by the Yosys script on the left

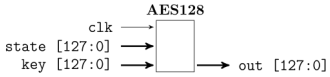

Fig. 2. A top-level block diagram of the original design

III. Demonstrative example

In this section, we provide a representative analysis of the AES-T800 benchmark[2], and demonstrate how we derive values for the fundamental Trojan properties. In the analysis, we use *Yosys* [17] for synthesis, *Icarus Verilog* [6] for simulation, and *NLTrace* [11] for waveform generation and trace analysis.

A. Specification

1) Original design: The functionality of the original design is specified such that the plaintext provided at the primary input *state* is encrypted using a synchronous hardware implementation of the AES encryption algorithm (controlled by the *clk* signal). Encrypted ciphertext is provided at the *out* primary output. Figure 2 shows a top-level block diagram of the original design.

2) Compromised design: The compromised design extends the specification of the original design as follows: "Once a predefined sequence of input plaintext is observed, the Trojan leaks the secret key from a cryptographic chip running the AES algorithm through a covert channel" (taken from the AES-T800 *Readme.txt*). Leaking data compromises data confidentiality, which satisfies the benchmark's fundamental *Maliciousness* property.

B. Design file structure

We start the analysis by accessing the *AES-T800* benchmark's root directory, which holds the source files for the Trojan benchmark. As for nearly all benchmarks, the root directory contains a PDF file that documents the benchmark specification (*AES-T800.pdf*), a text version of the specification in a *Readme.txt*, and a *src* directory, that holds the source code for the original design (*TjFree*), and the compromised design (*TjIn*). By comparing the two versions of the design, we notice that the two sets of files are not identical. There are five additional files in the compromised design: *lfsr.v*, *tbTOP.v*, *top.v*, *Trojan_Trigger.v*, *TSC v*. One file is missing: *simulation.do*. The inconsistencies between the original and the compromised design files violate the benchmark's fundamental *Stealthiness* property, because an IP integrator may immediately notice them.

C. Top-level design structure

We process the compromised design with the synthesis tool and execute the commands given in the left part of Figure 1, which generates the top-level block diagram given in the right part of Figure 1. When we compare the block diagram of the compromised design with the original design, we notice that the original top module is replaced by a new top module (implemented in *top.v*). This top module instantiates the original design, as well as two additional modules *Trigger* and *Trojan*. Further, the benchmark introduces one additional primary input *rst*, and one additional primary output *Capacitance*. We infer that to actually mount the attack, the attacker expects the IP integrator to instantiate the malicious *top* module, instead of the *aes_128* module as specified by the vendor of the original design, and to connect additional primary input/output ports. The IP integrator may immediately notice this deviation from the vendor specification, which would violate the benchmark's *Stealthiness* property.

[2]https://trust-hub.org/downloads/resource/benchmarks/AES/AES-T800.zip

```
$ tree TjFree
TjFree
├── aes_128.v
├── Read_Me.txt
├── round.v
├── simulation.do
├── table.v
└── test_aes_128.v

0 directories, 6 files
```

```
$ tree TjIn
TjIn
├── aes_128.v
├── lfsr.v
├── Read_Me.txt
├── round.v
├── table.v
├── tbTOP.v
├── test_aes_128.v
├── top.v
├── Trojan_Trigger.v
└── TSC.v

0 directories, 10 files
```

Fig. 3. Files of the original (TjFree) and compromised (TjIn) versions of the AES-T800 benchmark

D. Functional behavior

We investigate the benchmark's functional behavior by simulating the compromised design with the provided testbench, implemented in the file *tbTop.v*. We do not modify the testbench, except to specify a filename for dumping simulation results, and a directive to instruct the simulator to actually dump the results for the instantiated unit under test. We add the following two lines to the beginning of the `initial` block immediately after the instantiation of `uut`.

```
$dumpfile('VCDFILE);
$dumpvars(0,uut);
```

The testbench applies the trigger sequence to activate malicious payload (which is information leakage over a covert channel). As we learn from the testbench, the trigger sequence consists of four 128-bit words, applied to the top-level module's primary *state* input (which is the plaintext input of the AES cryptographic core), in the following order:

1) 128'h3243f6a8_885a308d_313198a2_e0370734,
2) 128'h00112233_44556677_8899aabb_ccddeeff,
3) 128'h0,
4) 128'h1

```
iverilog \
    -D 'VCDFILE="pre-synth.vcd"' \
    -o pre-synth \
    aes_128.v \
    lfsr.v \
    round.v \
    table.v \
    tbTOP.v \
    top.v \
    Trojan_Trigger.v \
    TSC.v
vvp pre-synth
```

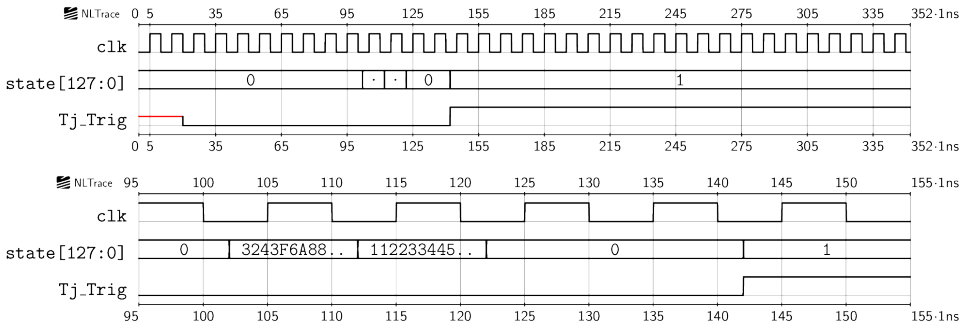

Fig. 4. Pre-synthesis simulation script and results

```
iverilog \
    -D 'VCDFILE="post-synth.vcd"' \
    -o post-synth \
    aes-t800_synth.v \
    tbTOP.v
vvp post-synth
```

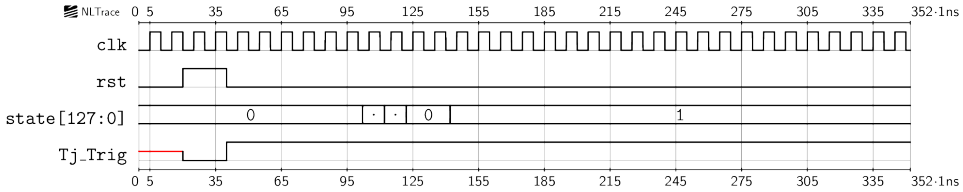

Fig. 5. Post-synthesis simulation script and results

```
cd /pre-synth.vcd:tbTOP/uut/Trojan
tree Tj_Trig load[63:0]
Tj_Trig
├──      0: X
├──  20000: 0
└── 142000: 1

load[63:0]
├──      0: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
├──  25000: 00000000000000000000000000000000000000000000000000000000000000000
├──  45000: 00000000000000001111111111111111000000000000000000000000011111111
├── 105000: 00000000000000000000000000000001111111111111111000000001111111
├── 115000: 00000000000000001111111111111111111111111111111111111111100000000
├── 125000: 00000000000000011111111111111110000000000000000000000001111111
├── 135000: 00000000000000011111111111111110000000000000000000000000000000
├── 145000: 00000000000000011111111111111110000000000000000000000000011111111
├── 155000: 00000000000000000000000011111111111111110000000000000000000000000
[...]
```

```
cd /post-synth.vcd:tbTOP/uut/Trojan
tree Tj_Trig load[63:0]
Tj_Trig
├──      0: X
├── 20000: 0
└── 40000: 1

load[63:0]
├──      0: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
├──  25000: 00000000000000000000000000000000000000000000000000000000000000000
├──  45000: 00000000000000001111111111111111000000000000000000000000011111111
├──  55000: 00000000000000000000000011111111111111110000000000000000000000000
├──  65000: 00000000000000000000000000000001111111111111111000000000000000
├──  75000: 00000000000000000000000000000000000000011111111111111100000000
├──  85000: 00000000000000000000000000000000000000000000000011111111111111
├──  95000: 11111111000000000000000000000000000000000000000000000001111111
├── 105000: 11111111111111111111111111111111111111111111111000000000000000
[...]
```

Fig. 6. Pre-synthesis (left) and post-synthesis (right) traces for the output of the payload's side-channel, provided by NLTrace

1) Pre-synthesis simulation: We simulate the HDL model of the compromised design using the shell script given on the left side in Figure 4. The waveform diagrams of the simulation results are given on the right side in Figure 4. There are two waveform diagrams: The upper one shows the trigger sequence for the entire simulated time, the lower one shows a magnified time frame to make the values of the *state* signal more readable. All value changes of *state* are within this time frame. We notice that the *Tj_Trig* trigger signal (see also Figure 1) goes from 0 to 1 after the trigger sequence is provided on the *state* input. We conclude that the RTL model seems to behave as specified.

2) Post-synthesis simulation: Now, let us synthesize the benchmark; it is supposed to be a **hardware** Trojan, so we may be well-advised to synthesize and again verify it. We synthesize the benchmark with the Yosys commands given in Figure 1, and write the synthesized benchmark into a Verilog file *aes-t800_synth.v* with the following Yosys command:

```
write_verilog aes-t800_synth.v
```

The synthesis tool generates log output for the trigger circuit as shown in Listing 1. We use the shell script given on the left side in Figure 5 to simulate the synthesized RTL model of the compromised design. The waveform diagram of the simulation results is given on the right side in Figure 5. What we notice is very interesting: The trigger signal still goes high, but it seems that it is not triggered by the rare input sequence as in Figure 4. Instead, *Tj_Trig* goes high immediately after *rst* goes low. It seems that the trigger circuit is not effective, because it seems to be always high. If so, any payload controlled by the trigger signal would be always active, thus being susceptible to detection during functional verification. This would violate the benchmark's *Stealthiness* property, because malicious payload is now observable without providing the trigger sequence. Because the trigger violates its specification, it violates the benchmark's *Correctness* property. It seems that synthesis removed the trigger behavior that was still present just now. If so, the benchmark's *Persistence* property may be violated as well.

The impact of unspecified trigger behavior is visible in the payload as well, which is implemented in two modules *lfsr_counter* and *TSC*. The *lfsr_counter* module implements a linear-feedback shift register (LFSR), used as a pseudo-random number generator (PRNG) to create a code-division multiple access (CDMA) code sequence [10] in the *TSC* module. Figure 7 shows the LFSR's output trace dumped by *NLTrace*, on the left side for the pre-synthesis simulation, right side for the post-synthesis simulation. Apart from the time shift caused by the different trigger behavior, the traces are identical and seem correct. A similar trace is given for the Trojan's (extra) *load* output which shows this shift as well. To validate the effectiveness of the data leakage circuit, an experimental setup to measure the leaked data and to demodulate the leaked key bits would be needed. We did not perform this kind of experiment, because it seems that the attack depends on the IP integrator to connect the *Capacitance* output, which may happen very unlikely. If left unconnected, not only the

```
cd /pre-synth.vcd:tbTOP/uut/Trojan/lfsr
tree Tj_Trig lfsr_stream[19:0]
Tj_Trig
├──      0: X
├──  20000: 0
└── 142000: 1

lfsr_stream[19:0]
├──      0: XXXXXXXXXXXXXXXXXXX
├──  25000: 00000011000000110001
├── 145000: 10000001100000011000
├── 155000: 11000000110000001100
├── 165000: 11100000011000000110
├── 175000: 01110000001100000011
├── 185000: 10111000000110000001
├── 195000: 11011100000011000000
├── 205000: 01101110000001100000
└── [...]
```

```
cd /post-synth.vcd:tbTOP/uut/Trojan/lfsr
tree Tj_Trig lfsr_stream[19:0]
Tj_Trig
├──     0: X
├── 20000: 0
└── 40000: 1

lfsr_stream[19:0]
├──      0: XXXXXXXXXXXXXXXXXXX
├──  25000: 00000011000000110001
├──  45000: 10000001100000011000
├──  55000: 11000000110000001100
├──  65000: 11100000011000000110
├──  75000: 01110000001100000011
├──  85000: 10111000000110000001
├──  95000: 11011100000011000000
├── 105000: 01101110000001100000
└── [...]
```

Fig. 7. Pre-synthesis (left) and post-synthesis (right) traces for the output of the payload's LFSR, provided by NLTrace

Listing 1. Synthesis log output for the trigger

```
yosys> read_verilog Trojan_Trigger.v
1. Executing Verilog-2005 frontend: Trojan_Trigger.v
Parsing Verilog input from 'Trojan_Trigger.v' to AST representation.
Generating RTLIL representation for module '\Trojan_Trigger'.
Note: Assuming pure combinatorial block at Trojan_Trigger.v:30.2-46.5 in
compliance with IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002. Recommending
use of @* instead of @(...) for better match of synthesis and simulation.
Note: Assuming pure combinatorial block at Trojan_Trigger.v:48.2-51.5 in
compliance with IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002. Recommending
use of @* instead of @(...) for better match of synthesis and simulation.
Successfully finished Verilog frontend

yosys> proc

2. Executing PROC pass (convert processes to netlists).
[...]
2.7. Executing PROC_DLATCH pass (convert process syncs to latches).
No latch inferred for signal '\Trojan_Trigger.\Tj_Trig' from process '\
        Trojan_Trigger.$proc$Trojan_Trigger.v:50$13'.
Latch inferred for signal '\Trojan_Trigger.\State0' from process '\
        Trojan_Trigger.$proc$Trojan_Trigger.v:30$1': $auto$proc_dlatch.cc:427:
        proc_dlatch$119
Latch inferred for signal '\Trojan_Trigger.\State1' from process '\
        Trojan_Trigger.$proc$Trojan_Trigger.v:30$1': $auto$proc_dlatch.cc:427:
        proc_dlatch$146
Latch inferred for signal '\Trojan_Trigger.\State2' from process '\
        Trojan_Trigger.$proc$Trojan_Trigger.v:30$1': $auto$proc_dlatch.cc:427:
        proc_dlatch$201
Latch inferred for signal '\Trojan_Trigger.\State3' from process '\
        Trojan_Trigger.$proc$Trojan_Trigger.v:30$1': $auto$proc_dlatch.cc:427:
        proc_dlatch$288
```

Trojan payload would be optimized away, but the entire malicious circuit (see the block diagram in Figure 1), which may violate the benchmark's *Persistence* property.

E. Code review

1) Trigger: Listing 2 shows the Verilog RTL implementation of the trigger circuit; There are two combinational `always` blocks (the first one starting at Line 30, the second one starting at Line 48). The first block is supposed to monitor the *state* input for the trigger sequence. The second block is supposed to activate the trigger signal *Tj_Trig* once the trigger sequence fully appeared on the *state* input. Let us remember the log output generated by the synthesis tool (Listing 1), and the unexpected post-synthesis trigger behavior; We conclude that this implementation of trigger behavior may not synthesize to a sequential circuit utilizing (edge-sensitive) flip-flops, but to a sequential circuit utilizing (level-sensitive) latches. A pre-/post-synthesis simulation mismatch seems to occur with the trigger's current implementation; it works in simulation, but not after synthesis. Analyzing the resources used by the trigger circuit seems to confirm that four latches are inferred instead of flip-flops (execute immediately after Line 1 of Listing 1: `proc; clean; opt -full; stat`). This is not specific to Yosys; The Verilog® register transfer level synthesis standard defines for any synthesis tool to infer a latch for a combinational block to hold the value of an output if it is not defined for all combinations of all input values [7, p. 17]. Latches (i.e., level-sensitive sequential models) "*are generally less predictable than edge-sensitive models due to the asynchronous nature of the signal interactions*" [7, p. 13]. The use of latches in synchronous digital designs is therefore discouraged. In many cases, latch inference results from buggy RTL code. Inferred latches make the trigger circuit susceptible to detection, because latches are actively searched for in routine verification tasks (due to above reasons). This may violate the benchmark's *Stealthiness* property (along with mixed tab/space indentation and DOS/Unix line endings), all the more because the infested design itself is a synchronous design, sensitive to the positive edge of the *clk* signal. It seems that we just confirmed the assumptions we made in Section III-D2: Because the trigger implementation may not meet its specification due to unpredictable behavior introduced by inferred latches, the benchmark's *Correctness* property is violated. Because trigger functionality disappears, the *Persistence* property is violated.

Listing 2. The trigger implemented in file *Trojan_Trigger.v*

```
21 module Trojan_Trigger(
22     input rst,
23     input [127:0] state,
24     output Tj_Trig
25     );
26
27     reg Tj_Trig;
28     reg State0, State1, State2, State3;
29
30     always @(rst, state)
31     begin
32         if (rst == 1) begin
33             State0 <= 0;
34             State1 <= 0;
35             State2 <= 0;
36             State3 <= 0;
37         end else if (state == 128'h3243f6a8_885a308d_313198a2_e0370734) begin
38             State0 <= 1;
39         end else if ((state == 128'h00112233_44556677_8899aabb_ccddeeff) && (
                State0 == 1)) begin
40             State1 <= 1;
41         end else if ((state == 128'h0) && (State1 == 1)) begin
42             State2 <= 1;
43         end else if ((state == 128'h1) && (State2 == 1)) begin
44             State3 <= 1;
45         end
46     end
47
48     always @(State0, State1, State2, State3)
49     begin
50         Tj_Trig <= State0 & State1 & State2 & State3;
51     end
52
53 endmodule
```

Listing 3. The payload implemented in file *TSC.v*

```
21 module TSC(
22     input rst,
23     input clk,
24      input Tj_Trig,
25     input [127:0] key,
26      output [63:0] load
27     );
28
29     reg [63:0] load;
30     wire [19: 0] counter;
31
32     lfsr_counter lfsr (rst, clk, Tj_Trig, counter);
33     always @ (posedge clk)
34         begin
35             if (rst == 1) begin
36                 load <= 0;
37             end else begin
38                 load[0] <= key[0] ^ counter[0];
39                 load[1] <= key[0] ^ counter[0];
40                 load[2] <= key[0] ^ counter[0];
41 [...]
```

2) Payload: Malicious payload leaks the least significant byte of the secret key over a side-channel, and is implemented in two modules as shown in Listing 3 and Listing 4. The *lfsr_counter* module implements an LFSR, used as a PRNG to create a CDMA code sequence in the *TSC* module as proposed by Lin et al. [10]. We notice that the code for implementing the side channel appears fairly bloated (112 lines of code that come with mixed indentation and line endings), which may be implemented more efficiently. The amount of payload code may violate the benchmark's *Stealthiness* property.

Listing 4. The payload implemented in file *lfsr.v*

```
1
2  // It implements X^20 + X^13 + X^9 + X^5 + 1
3  module lfsr_counter (
4      input rst, clk, Tj_Trig,
5     output [19:0] lfsr
6      );
7
8      reg [19:0] lfsr_stream;
9      wire d0;
10
11
12     assign lfsr = lfsr_stream;
13     assign d0 = lfsr_stream[15] ^ lfsr_stream[11] ^ lfsr_stream[7] ^
               lfsr_stream[0];
14
15     always @(posedge clk)
16         if (rst == 1) begin
17             lfsr_stream <= "10011001100110011001";
18         end else begin
19             if (Tj_Trig == 1) begin
20                 lfsr_stream <= {d0,lfsr_stream[19:1]};
21             end
22         end
23
24 endmodule
```

F. Summary

We provide an in-depth analysis of the TrusHUB *AES-T800* benchmark. We analyze the design's file structure, and learn that the sets of files are not equal for the original and the compromised designs. When analyzing the top-level design hierarchy, we notice that extra modules, primary inputs and outputs are added. When performing pre-synthesis simulation, the benchmark works as expected; but once RTL synthesis is performed, the post-synthesis simulation yields unexpected results, rendering the trigger ineffective. When reviewing the benchmark's source code, we find out that the trigger implementation inferres latches instead of flip-flops, rendering the trigger circuit ineffective. The attack model requires the IP integrator to connect an extra payload output, which is very unlikely to happen, and leaves malicious circuitry unconnected. As a result, the entire malicious circuit may be optimized away.

All of the above may violate the benchmark's *Correctness*, *Stealthiness*, and *Persistence* properties. As a consequence, the benchmark's *Effectiveness* property may not be satisfied; Although the benchmark's *Maliciousness* property holds, the benchmark may be considered ineffective, because specified malicious behavior is not correctly implemented, it may become visible at various levels, and it may disappear on one or the other occasion.

IV. Results

A. Experimental setup

1) Verilog: We synthesize Verilog designs with *Yosys* (version 0.22+4), and simulate them with *Icarus Verilog* (version 11.0), pre- and post-synthesis. In order to avoid biased results due to a limited set of tools, we use *Xilinx Vivado* (version 2022.2) to synthesize the *PIC16F84* benchmarks, and *Xilinx xsim* (version 2022.2) to simulate the designs, with stimuli provided by a testbench written in VHDL.

2) VHDL: We synthesize VHDL designs with the commercial version of Yosys included in the YosysHQ Tabby CAD Suite (version 20221011) by using the *Verific* frontend. We simulate VHDL designs with *GHDL* (version 1.0.0), pre-synthesis. Synthesized VHDL designs are written to a Verilog netlist.

3) Technology netlists: Benchmarks that are provided as technology-mapped Verilog netlists are mapped to the *Synopsys SAED 90nm* digital standard cell library. In order to make these benchmarks accessible to Yosys, we implemented a cell library that maps SAED standard cell types to RTL models as specified in the *Digital Standard Cell Library Databook* [2]. When synthesized with Yosys, the standard cells are mapped to Yosys internal cell types.

4) A note on persistence: We do not map the benchmarks to a specific target technology in order to determine their *Persistence* property. Once a benchmark survives RTL synthesis, we consider its *Persistence* property satisfied (in this study). However, this does not necessarily imply that malicious functionality is present in hardware, but must be specifically evaluated for any potential target technology.

B. Experimental results

Table II summarizes the findings of our study. We learn that there are three benchmark designs that model effective hardware Trojans: *BasicRSA-T100*, *memctrl-T100*, and *wb_connmax-T300*. These Trojans effectively hide their injected functionality, and model malicious behavior that can be exploited by the attacker. These benchmarks can serve as role models for effective benchmark design, particularly *memctrl-T100*, which implements a combined hardware/software attack. However, Table II also shows that the TrustHUB benchmarks entail considerable (undocumented) limitations. Only three out of 83 studied designs are usable as is (i.e., 3.6%). The majority of benchmark designs may require debugging to become effective as defined in Section II-D. In the following, we summarize the most important results, and identify some pointers for debugging the benchmarks:

1) Pre-/Post-synthesis simulation mismatch: Many benchmarks are applicable to a limited extent, because malicious functionality only exists in (pre-synthesis) RTL simulation. Once the design is synthesized, malicious functionality disappears due to pre-/post-synthesis simulation mismatch. For instance, this is the case for many of the *AES* benchmarks; and also for all of the *PIC16F84* designs, which use a counter as a trigger that is implemented using level-sensitive sequential logic (instead of edge-sensitive sequential logic), yielding unexpected behavior after synthesis.

2) Unsatisfiable trigger conditions: Most of the benchmarks provided as technology netlists, such as *s15850, s35932, s38417, s38584, b15, b19, vga_lcd*, and *wb_connmax*, employ triggers which were automatically inserted by *design vulnerabilty analysis* into ultra-low activity regions of the circuits, connecting the trigger inputs to untestable redundant nets. "*Untestable redundant nets are not testable because they are masked by a redundant logic, and they are not observable through primary output or scan cells.*" [12] While this may sound beneficial in the first moment, this approach makes it hard for the attacker to actually control the Trojan, and no guarantee is made that the trigger condition is satisfiable at all. These "Trojans" are therefore not testable, not usable, and may be considered ineffective. Likewise, we were not able to verify correct functional behavior of benchmarks *RS232* T1000-T2000. A testbench that exhibits malicious behavior would greatly improve a benchmark's verifiability. We tried with a SAT solver to find valuations of the trigger inputs such that the trigger output evaluates to *True*, but we were unsuccessful in almost all cases – the triggers simply were not triggerable. For instance, it was not possible to activate the trigger signal for the *wb_conmax-T100* Trojan.[3] Documentation on how to trigger malicious functionality under a given threat model would help a lot to understand these kinds of Trojans.

3) Unconnected output ports: Malicious functionality is optimized away for some benchmarks during routine optimization passes, e.g., due to disconnected signals, or additional primary output ports, which would not be connected in a real setting. Many of the *AES* benchmarks are rendered ineffective because of this.

[3]https://trust-hub.org/downloads/resource/benchmarks/WB_CONMAX/wb_conmax-T100.rar

4) Additional resources: All of the *AES* benchmarks introduce additional files, modules, and/or extra primary input/output ports, which may be easily detectable in a real attack involving such a Trojan.

5) Incorrect original design: For all of the *RS232* RTL benchmarks (T100–T901, T2100–T2400), the original design does not correctly implement its specification, such that it is impossible to send data over the UART; note that benchmarks T100–T901 correctly implement malicious functionality, which remains stealthy and persistent. For the *MC8051* benchmarks, a synthesizable read-only memory (ROM) is not available; note that most of these benchmarks survive RTL synthesis, although latches are inferred.

6) Unspecified malicious behavior: Benchmarks *RS232* T2100–T2400 violate the compromised design's specification because they implement malicious behavior which is not specified: In addition to setting signal *rec_readyH* to 0 upon activation (which is specified), also *rec_dataH* is set to 8'd0 (which is unspecified).

7) Non-malicious behavior: For benchmarks *AES-T2300–T2800*, we do not observe malicious behavior. While injected functionality violates the original specification by XORing the least-significant bit (LSB) of ciphertext, we cannot determine the potential attacker's intention to carry out such attacks.

8) Ineffective triggers: Most of the *AES-T2300–T2800* benchmarks employ triggers which monitor rare internal signals, which however toggle multiple times even for a small number of test cases. Likewise, there are triggers implemented as 4-bit counters, which may not able to successfully hide injected functionality during functional verification. Benchmarks *AES-T2500* to *AES-T2800* implement sequential trigger circuits that are controlled by asynchronous reset functionality that is level-sensitive to an external signal and therefore represents the real "trigger" in these benchmarks.

9) Buggy wiring: Many of the benchmarks provided as technology netlists connect the two inputs of one of the trigger circuit's AND gates to the same input signal, while leaving the output signal *Tj_OUT2* disconnected.

TABLE II
THE RESULTS OF OUR STUDY ON FUNDAMENTAL TROJAN PROPERTIES OF THE TRUSTHUB BENCHMARK SUITE (C: CORRECTNESS, M: MALICIOUSNESS, S: STEALTHINESS, P: PERSISTENCE, E: EFFECTIVENESS). ● MEANS *True*, ○ MEANS *False*

	C	M	S	P	E
AES					
T100	○	●	○	○	○
T200	○	●	○	○	○
T300	○	●	○	○	○
T400	○	●	○	○	○
T500	○	●	○	○	○
T600	○	●	○	○	○
T700	○	●	○	○	○
T800	○	●	○	○	○
T900	○	●	○	○	○
T1000	○	●	○	○	○
T1100	○	●	○	○	○
T1200	○	●	○	○	○
T1300	○	●	○	○	○
T1400	○	●	○	○	○
T1500	○	●	○	○	○
T1600	○	●	○	○	○
T1700	○	●	○	○	○
T1800	○	●	○	○	○
T1900	○	●	○	○	○
T2000	○	●	○	○	○
T2100	○	●	○	○	○
T2200	●	●	○	○	○
T2300	●	○	○	●	○
T2400	●	○	○	●	○
T2500	○	○	○	●	○
T2600	○	○	○	●	○
T2700	○	○	○	●	○
T2800	○	○	○	●	○
wb_conmax					
T100	○	●	●	●	○
T200	○	●	○	○	○
T300	●	●	●	●	●
s35932					
T100	○	●	●	●	○
T200	○	●	●	●	○
T300	○	●	○	●	○
s38417					
T100	○	●	●	●	○
T200	○	●	●	●	○
T300	○	●	○	●	○
s38584					
T100	○	●	●	●	○
T200	○	●	●	●	○
T300	○	●	●	●	○
vga_lcd					
T100	○	●	●	●	○

	C	M	S	P	E
RS232					
T100	○	●	●	●	○
T200	○	○	●	●	○
T300	○	●	●	●	○
T400	○	●	●	●	○
T500	○	●	●	●	○
T600	○	●	●	●	○
T700	○	●	●	●	○
T800	○	●	●	●	○
T900	○	●	●	●	○
T901	○	●	●	●	○
T1000	○	●	●	●	○
T1100	○	●	●	●	○
T1200	○	●	●	●	○
T1300	○	●	●	●	○
T1400	○	●	●	●	○
T1500	○	●	●	●	○
T1600	○	●	●	●	○
T1700	○	●	●	●	○
T1800	○	●	●	●	○
T1900	○	●	●	●	○
T2000	○	●	●	●	○
T2100	○	●	○	○	○
T2200	○	●	○	○	○
T2300	○	●	○	○	○
T2400	○	●	○	○	○
MC8051					
T200	○	●	○	○	○
T300	○	●	○	○	○
T400	○	●	○	○	○
T500	○	●	○	○	○
T600	○	●	●	●	○
T700	○	●	○	●	○
T800	○	●	○	●	○
PIC16F84					
T100	○	●	○	○	○
T200	○	●	○	○	○
T300	○	●	○	○	○
T400	○	●	○	○	○
BasicRSA					
T100	●	●	●	●	●
T200	○	●	○	○	○
T300	●	●	○	●	○
T400	●	●	○	●	○
memctrl					
T100	●	●	●	●	●

V. CONCLUSIONS

The most-important conclusion of this study is the discovery of considerable undocumented limitations of the TrustHUB hardware Trojan benchmark suite, which render most of its benchmarks ineffective when attempting to use them in hardware. In order to satisfy intuitive expectations on hardware Trojan benchmarks to be effective in hardware, the TrustHUB benchmarks may require re-design which considers the fundamental Trojan properties proposed in Section II-C. Potentially ineffective benchmarks pose a severe problem to the trustworthiness of integrated circuits, because experiments performed on such benchmarks may potentially lead to wrong conclusions. All of the above suggests that the TrustHUB benchmarks may require either thorough review, or transparent documentation[4] of all limitations and assumptions made for each specific benchmark design. It may be necessary that some of the works using the TrustHUB benchmarks must revise their experiments, and repeat them with improved versions of the benchmarks.

We generalize from the findings of our study to the following guidelines for effective benchmark design: (1) Obey the threat model, (2) Avoid exposure to user interfaces (e.g., by introducing additional files, modules, primary inputs and outputs, etc.), (3) Obey good design practice, (4) Keep malicious code short and simple, (5) Make sure nothing essential is optimized away, (6) Avoid pre-/post-synthesis simulation mismatch to make sure the benchmark works in hardware, (7) Provide testbenches to ensure the benchmark's verifiability, (8) (Some) black-box benchmarks may increase expressiveness of tests.

[4]Detailed reports including scripts will be released as a series of blog posts after the conference at **https://chipsecs.com/r/rotth**

ACKNOWLEDGMENTS

The author would like to thank his students (in alphabetical order) for their valuable input and their help in conducting experiments: Rainer Bayr, Felix Braun, David Breuss, Florian Egert, Josef Gull, Johannes Hiesberger, Severin Jäger, Maximilian Prindl, Stefan Riesenberger, Florian Schuller, Lukas Vogl, Philipp-Sebastian Vogt. Thank you!

References

[1] R. S. Chakraborty, S. Narasimhan, and S. Bhunia. "Hardware Trojan: Threats and emerging solutions". In: *2009 IEEE International High Level Design Validation and Test Workshop*. 2009, pp. 166–171. DOI: 10.1109/HLDVT.2009.5340158.

[2] *Digital Standard Cell Library SAED_EDK90_CORE Databook*. URL: https://web.engr.oregonstate.edu/~traylor/ece474/reading/SAED_Cell_Lib_Rev1_4_20_1.pdf (visited on 08/18/2023).

[3] Pravin Gaikwad, Jonathan Cruz, Prabuddha Chakraborty, Swarup Bhunia, and Tamzidul Hoque. "Hardware IP Assurance against Trojan Attacks with Machine Learning and Post-Processing". In: *J. Emerg. Technol. Comput. Syst.* 19.3 (June 2023). ISSN: 1550-4832. DOI: 10.1145/3592795. URL: https://doi.org/10.1145/3592795.

[4] Syed Kamran Haider, Chenglu Jin, Masab Ahmad, Devu Shila, Omer Khan, and Marten van Dijk. "Advancing the state-of-the-art in hardware trojans detection". In: *IEEE Transactions on Dependable and Secure Computing* (2017).

[5] Wei Hu, Chip-Hong Chang, Anirban Sengupta, Swarup Bhunia, Ryan Kastner, and Hai Li. "An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2021), pp. 1010–1038. DOI: 10.1109/TCAD.2020.3047976.

[6] *Icarus Verilog*. URL: https://steveicarus.github.io/iverilog/ (visited on 05/22/2023).

[7] "IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis". In: *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1* (2005), pp. 1–116. DOI: 10.1109/IEEESTD.2005.339572.

[8] Christian Krieg, Claire Wolf, and Axel Jantsch. "Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7827620.

[9] Christian Krieg, Claire Wolf, Axel Jantsch, and Tanja Zseby. "Toggle MUX: How X-Optimism Can Lead to Malicious Hardware". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC'17. Austin, TX, USA: Association for Computing Machinery, 2017. ISBN: 9781450349277. DOI: 10.1145/3061639.3062328. URL: https://doi.org/10.1145/3061639.3062328.

[10] Lang Lin, W. Burleson, and C. Paar. "MOLES: Malicious off-chip leakage enabled by side-channels". In: *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*. 2009, pp. 117–122. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5361303.

[11] *NLTrace – Analyze traces*. A web application is available at https://nltrace.io. URL: https://nltrace.org (visited on 05/22/2023).

[12] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. "On design vulnerability analysis and trust benchmarks development". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 471–474. DOI: 10.1109/ICCD.2013.6657085.

[13] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. "Benchmarking of Hardware Trojans and Maliciously Affected Circuits". In: *Journal of Hardware and Systems Security* 1.1 (Mar. 2017), pp. 85–102. ISSN: 2509-3436. URL: https://doi.org/10.1007/s41635-017-0001-6.

[14] Mohammad Tehranipoor and Farinaz Koushanfar. "A Survey of Hardware Trojan Taxonomy and Detection". In: *IEEE Design & Test of Computers* 27.1 (2010), pp. 10–25. DOI: 10.1109/MDT.2010.7.

[15] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. "FANCI: identification of stealthy malicious logic using boolean functional analysis". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 697–708.

[16] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. "Hardware Trojans: Lessons Learned after One Decade of Research". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (May 2016). ISSN: 1084-4309. DOI: 10.1145/2906147. URL: https://doi.org/10.1145/2906147.

[17] *Yosys Open SYnthesis Suite*. URL: https://yosyshq.net/yosys/documentation.html (visited on 05/22/2023).

[18] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. "VeriTrust: Verification for hardware trust". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.7 (2015), pp. 1148–1161.