HaTCh: Hardware Trojan Catcher

Syed Kamran Haider[†], Chenglu Jin[†], Masab Ahmad[†], Devu Manikantan Shila[‡], Omer Khan[†] and Marten van Dijk[†]

[†]University of Connecticut –
{syed.haider, chenglu.jin, masab.ahmad, khan}@uconn.edu, vandijk@engr.uconn.edu
[‡]United Technologies Research Center – manikad@utrc.utc.com

November 16, 2014

Abstract

With the growing trend of repeated reuse of existing design components, use of third party IP cores has become a common practice in Electronic Design Automation (EDA) industry. However third party IP cores are known to have potential malwares or hardware trojans which could compromise the security of the whole system.

We provide a formal classification of possible hardware trojans according to their different properties and analyze in detail the class coined *XX-St-D-F*, which is the collection of trojans that (1) use *St*andard I/O channels to deliver malicious payload, (2) are embedded in an IP core with *D*eterministic functionality, and (3) are designed to violate the *F*unctional specification. We provide a hierarchy of subclasses $H_{t,1} \subseteq H_{t,2} \subseteq \ldots \subseteq H_{t,d} \subseteq \ldots \subseteq \bigcup_{d\geq 1} H_{t,d} = H_t \subseteq \bigcup_{t\geq 0} H_t =$*XX-St-D-F*, where t indicates the number of clock cycles between "activating/triggering" the trojan and the moment a malicious payload is delivered and where d stands for the number of wires involved in the "trigger signal". We show that all *XX-St-D-F* trojans benchmarked by Trusthub [1] belong to $H_{t,1}$ and we design new trojans that belong to each of the classes $H_{t,d}$. This demonstrates that the currently found/benchmarked trojans are very likely only the tip of the iceberg.

By using the synthesized netlist description of an IP core as input, we introduce a new tool called HaTCh which learns in a precomputation or "learning" phase how to add additional "tagging" circuitry to the IP core such that, as soon as an embedded *XX-St-D-F* trojan is triggered, the tagging circuitry raises an exception to prevent the trojan from manifesting malicious behavior. We show that HaTCh parameterized for $H_{t,d}$ has zero false negatives among trojans in $H_{t,d}$ and depending on the duration of the precomputation (learning phase) the false positive rate can be designed to approach zero. For a sample among the Trusthub [1] benchmarks in *XX-St-D-F*, we show a false positive rate of 10^{-5} (for $d = 1$).

The learning phase of HaTCh has a complexity of $O(d2^d \cdot |Core| \ln |Core|)$ where $|Core|$ is the number of wires in the IP core. We show how this complexity can potentially be reduced by considering "interesting" subsets of $H_{t,d}$, one of which leads to $O(|Core|^3)$ complexity independent of d. The tagging circuitry for our sample of benchmarks has area overhead from 0.02% to 7.63% for non-pipelined tagging circuitry, and from 0.02% to 15.25% for pipelined tagging circuitry which is useful for designs having strict timing constraints.

Contents

1 Introduction **3**
- 1.1 Contributions . . . 5

2 Related Work **6**

3 Background & Adversarial Model **7**
- 3.1 IP Core Design Space . . . 7
  - 3.1.1 Hardware Trojan . . . 7
  - 3.1.2 An Exploit . . . 8
- 3.2 IP Core Design Flow & Adversarial Model . . . 8

4 Classification of Hardware Trojans **9**

5 Formal Hardware Trojan Framework **10**
- 5.1 IP Core . . . 11
- 5.2 Specification . . . 13
  - 5.2.1 CoreSim . . . 13
  - 5.2.2 OutSpec . . . 14
- 5.3 *XX-St* Trojan Behavior . . . 16

6 Hardware Trojan Class $H_{t,d}$ **16**
- 6.1 $H_{0,1}$ Class Trojan Example . . . 18
- 6.2 $H_{0,2}$ Class Trojan Example . . . 18
- 6.3 $H_{0,d}$ Class Trojan Example . . . 19

7 HaTCh Framework **20**
- 7.1 Learning Phase . . . 20
  - 7.1.1 Zero False Negatives . . . 21
  - 7.1.2 Non-Zero False Positives . . . 22
- 7.2 Tagging Phase . . . 24
  - 7.2.1 Pipelined Logic . . . 24
  - 7.2.2 Area Overhead . . . 25
- 7.3 HaTCh Process Example . . . 25

8 Evaluation **25**
- 8.1 Trusthub Benchmark Suite . . . 26
  - 8.1.1 XX-St-D-F Trojans . . . 26
  - 8.1.2 TA-Si and TA-St-D-NF Trojans . . . 26
  - 8.1.3 AA-Si and AA-St-D-NF Trojans . . . 27
- 8.2 Experimental Setup & Methodology . . . 28
  - 8.2.1 RTL Synthesis . . . 28
  - 8.2.2 Simulation . . . 28
  - 8.2.3 HaTCh Learning and Tagging Phases . . . 28
  - 8.2.4 HaTCh Optimizations . . . 28
- 8.3 Experimental Results . . . 29

8.3.1 s-Series Benchmarks . . . 29
8.3.2 Other benchmarks . . . 29
8.4 Area Overhead . . . 30
8.5 Techniques to reduce HaTCh Complexity . . . 31
8.5.1 Golden Input Patterns . . . 31
8.5.2 Modular Approach for IP cores . . . 31
8.6 Techniques to reduce HaTCh Area Overhead . . . 31
8.6.1 Formal Proofs to reduce the blacklist . . . 31
8.6.2 Clustering approach for high level HaTCh . . . 31

9 Conclusion **31**

A Analysis of *k*-*XOR-LFSR* **34**
A.1 Classification . . . 34
A.2 Improving HaTCh . . . 35

B A Counter-Based $H_{0,2}$ Trojan Example **35**
B.1 Detection by HaTCh with $d = 2$ . . . 36

1 Introduction

Modern electronic systems heavily use third party IP (intellectual property) cores as their basic building blocks [2]. An IP core is a reusable block of logic, tailored to perform a particular operation in an efficient manner, that is an intellectual property of one party. Optimized for area and performance, the IP cores are essential elements of design reuse in electronic design automation (EDA) industry and save a lot of resources to redesign the components from scratch. UARTs, DSP units, Ethernet controllers and PCI interfaces etc. are some examples of the IP cores [3].

IP cores are broadly categorized in hard and soft IP cores. Hard IP cores are transistor level representations of the core's functionality. Soft IP cores are offered either in a hardware description language, such as Verilog or VHDL, as synthesizable RTL or as generic gate-level netlists which is a boolean function representation of core's logic.

Both hard IP cores and soft IP cores offered as gate-level netlists are usually called 'closed source', as their high level RTL source code is not provided. Such cores give their vendors significant protection against reverse engineering of the cores by obfuscating the algorithmic and implementation tricks. To even improve the strength of obfuscation in the near future, indistinguishability obfuscators[1] $i\mathcal{O}$ (for polynomial-size circuits), as recently developed in the crypto community, may be used as soon as their constructions attain acceptable performance overheads [4] [5].

Even though (partially) obfuscated cores protect the intellectual property to certain extent, they give rise to a critical security problem: how to make sure that a third party closed source IP core does not embed a *Hardware Trojan*? The IP core vendor acting as an adversary could implant a malicious circuitry in the IP core, through its source code or by using some malicious tools to generate the core netlist, for privacy leakage or denial of service attacks [6]. More specifically, before using a closed source IP core, the following two questions must be addressed;

[1] For any two functionally equivalent circuits C_1 and C_2, the two distributions of $i\mathcal{O}(C_1)$ and $i\mathcal{O}(C_2)$ are computationally indistinguishable.

1. Is the IP core based on a **trusted source code**?

2. Has the IP core been generated using a **trusted toolchain**?

In order to address these questions, one needs to have a basic intuition about how does a hardware trojan work. A hardware trojan usually consists of two parts: a trigger circuitry which activates the trojan upon a rare condition or event called ***trigger condition***, and a payload circuitry which performs the malicious operation called 'payload' as intended by the adversary. The trigger condition manifests itself in the form of a boolean value of certain wires. The trigger circuitry is implemented semantically as a comparator which compares the values of these relevant wires with the desired trigger condition and outputs the result in the form of a boolean value on another wire $Trig$ which we call ***trigger signal*** (i.e. $Trig = 1$ upon trigger condition, $Trig = 0$ otherwise). The payload circuitry takes the trigger signal $Trig$ as input and performs malicious operation upon $Trig = 1$. The *trigger signal* must not be confused with *trigger condition*; trigger condition is an event which causes the trojan activation, whereas trigger signal is the output of trigger circuitry which signals the payload circuit to show malicious behavior. Notice that $Trig$ remains 0 until the trigger condition occurs, which is a very rare event, i.e. it remains an "unused" input to the next combinational logic to which it is connected. This leads to the basic principle of hardware trojans detection i.e. identifying the unused wires (i.e. the trigger signal) in the design.

We notice that here $|Trig| = d = 1$, i.e. the trigger signal consists of only one wire which takes a certain value (e.g. $Trig = 1$) *only* upon the trigger condition. We call d the dimension of the trojan which shows the width of the trigger signal. A trojan with $d = 1$ is relatively easy to detect since one out of $2^d = 2^1 = 2$ possible logic values of the trigger signal could be the result of a trigger condition. However, if for example $d = 2$, i.e. the trigger signal is a certain combination of two wires, then one out of $2^d = 2^2 = 4$ possible combinations could be the result of a trigger condition which makes it hard to detect such trojans because of the exponentially increased computational complexity.

This key observation of ours leads to one of the major contributions of this paper. We introduce a hierarchical model of hardware trojans, along with design examples, based on the number of wires involved (dimension d) in the trojan trigger signal which, to best of our knowledge, has never been done before. This model helps quantifying the complexity of the hardware trojans to better understand the required properties of the future techniques and tools for hardware trojan detection.

There exist several different trojan detection schemes in the literature which try to detect a potentially malicious circuit module in the IP core by identifying its trigger wire(s). Some of these state of the art techniques include UCI [7], VeriTrust [8] and FANCI [9]. These techniques, particularly VeriTrust and FANCI, were known to detect all benchmarked trojans with significantly low false positive rates until now when a most recent work called DeTrust [10] showed a trojan design methodology to bypass these protection schemes.

The key idea behind the trojan design methodology of DeTrust is that the trojan trigger circuit logic should be intermixed with the normal circuitry and spread over multiple sequential levels so that it appears to be a part of "normal circuitry". We notice that, in some cases, this leads to the hardware trojans for which the trigger signal does not boil down to only one wire, i.e. they have dimension $d > 1$. The existing techniques which only consider trojans with a trigger signal consisting of one wire (i.e. $d = 1$) cannot detect a trojan whose trigger signal is a certain combination of more than one wires ($d > 1$).

To bridge this gap, we introduce a powerful hardware trojan detection tool called ***Ha****rdware* ***T****rojan* ***C****atc****h****er* (HaTCh) which takes a parameter d' and detects all the trojans with $0 < d \leq d'$.

We prove HaTCh to have a zero false negative rate. Experimental results show a significantly low ($1/10^5$) false positive rate and an area overhead between 0.02% to 7.63% (non-pipelined) and 15.25% (pipelined) for a sample of Trusthub [1] benchmarks.

1.1 Contributions

Following are the main contributions of this paper:

1. A formal classification of hardware trojans, referred to as *{TA/AA}-{St/Si}-{D/ND}-{F/NF}*, based on their four different properties: activation mechanism, payload channels, determinism and payload functionality. We show how *XX-Si-XX-XX*, *XX-St-XX-NF* and *XX-St-ND-XX* can manifest malicious behavior without being detected, i.e. for which no tool with zero false negative rate exists.

2. We categorize *TA-St-D-F* into a hierarchy of hardware trojans $H_{t,d}$ based on dimension d which shows the number of wires in the trigger signal and depicts the stealthiness of the trojan, and t which shows the latency between trigger condition and malicious behavior.

3. We show that the existing *TA-St-D-F* trojan benchmarks are in $H_{t,1}$ in our hierarchy, which means that the currently found/benchmarked trojans are very likely only the tip of the iceberg: trojan designs can be improved for more stealthiness and so can the tools to detect them. In fact we introduce a new trojan design for trojans in $H_{0,d}$.

4. We present a tool HaTCh to detect $H_{t,d} \in$ *XX-St-D-F* with the following properties:

   - Zero false negatives
   - $1/T$ false positive rate that can be made to approach zero
   - Learning complexity $O(d2^d \cdot |Core| \ln(|Core| \cdot T))$
   - Produces tagging circuitry with small overhead

5. Our experimental results on a certain benchmark group show that HaTCh has a false positive rate of 10^{-5}. The optimizations implemented in HaTCh for area overhead lead to 4.5 times reduced area overhead leading to 4.18% (non-pipelined) and 8.34% (pipelined) overheads on average.

6. We discuss techniques to reduce the learning complexity of HaTCh e.g. by using linear invariant checking.

Rest of the paper is organized as follows; section 2 talks about the existing techniques for hardware trojan detection along with their limitations. Section 3 provides some basic background of hardware trojans. Section 4 presents a thorough classification of hardware trojans based on their activation mechanisms, payload channels and behavior of the IP core in which they are embedded. Section 5 provides a formal mathematical framework for a particular class of HW trojans, whereas section 6 introduces a new hierarchical model of this specific class. The details of the HaTCh tool and its experimental evaluation are in section 7 and 8 respectively and we conclude in section 9.

2 Related Work

Hardware trojans have recently gained significant interest in the security community [11], [12], [13]. The works [12] and [13] showed how malicious entities can exist in hardware, while Skorobogatov *et al.* [14] showed evidence of such backdoors in military grade devices. Nefarious designs have also been deployed and detected in wireless communications devices [15]. Recent works have mostly focused on detection [16] and identification schemes [17], which assess to what extent the pieces of hardware may be vulnerable, and how related trojans can be classified.

Hicks *et al.* [7] proposed to detect hardware trojans through *unused circuit identification* (UCI). Their solution centers on the fact that the hardware trojan circuitry will not be used within a design, and hence such minimally used logic can be distinguished from the design specification. However, due to functional verification constraints, whole designs cannot be analyzed in optimal time, and hence the scheme identifies large portions of the design as a potential hardware trojan. This results in a high false positive rate, and recent works by some papers have even succeeded in breaking this scheme [18], [19].

Veritrust [8] is another scheme proposed by Zhang *et al.* that identifies suspicious wires that seem redundant in comparison with the design. The scheme uses Karnaugh-maps, and excites portions of the circuit using the design specification, given the fact that the design spec will not activate the hardware trojan. However, the design spec may not activate all circuitry in the design, and the remaining wires are all classified as potential hardware trojans, and contribute heavily to the false positive rate.

Waksman *et al.* designed methods to apply boolean function based heuristics to flag suspicious wires in a design [9], stemming ideas from their previous work on hardware obfuscation [20]. This solution may be suitable for cases where backdoors are evident as wires in the design. However, the admitted weakness of this solution is that the scheme suffers from false positives, and is recently broken along with VeriTrust by *DeTrust* [10]. Moreover, this method is a probabilistic method which uses a threshold and some heuristics to determine if a wire should be considered suspicious. This could lead to a false negative where a trojan related wire is regarded as 'not' suspicious because of using low threshold to reduce false positives.

DeTrust [10] is the most recent scheme targeting hardware trojans. It develops several new stealthy hardware trojans whose circuitries are intermixed with the normal design. Therefore, by having trojan circuits being part of the normal design, previous schemes would designate them as non-malicious, resulting in a false negative rate. However, they only discuss on how to improve the current works (FANCI and VeriTrust) to detect their trojans. The paper also shows that FANCI exhibits a much higher false positive rate than expected.

Further works construct and detect hardware trojans through side channels [21], [22], [23]. Such hardware trojans remain implicitly on, and have no effect on the functionality of the circuit [24]. Side channels include power based channels [25], [26], as well as heat based channels [27] [28]. Power based trojans force the circuit to dissipate more and more power to either damage the circuit, or simply waste energy [29]. Heat based trojans leak important information via heat maps [30], where highs and lows in heat dissipation can be interpreted as 1's and 0's.

The above works use trojans from the TrustHub hardware trojan benchmarks suite, in which all trojans are explicitly triggered. This explicitness forgoes the lack of implicitness, due to which all the above schemes are able to detect the benchmarked trojans. These works do not cater for implicit higher dimensional ($d > 1$) trojans, and also do not provide a theoretical basis for their methods. This paper fills this gap by unifying trojan taxonomy, and providing a framework for

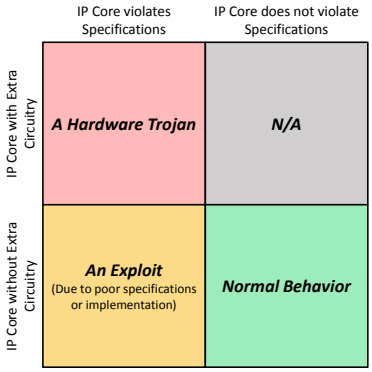


Figure 1: IP Core Design Space: An IP core violating the specifications and having an *extra* circuitry contains a Hardware Trojan whereas a core violating the specifications without having any extra circuitry contains an Exploit. Note that a core having extra circuitry and not violating the specifications is not applicable.

hardware trojan detection.

3 Background & Adversarial Model

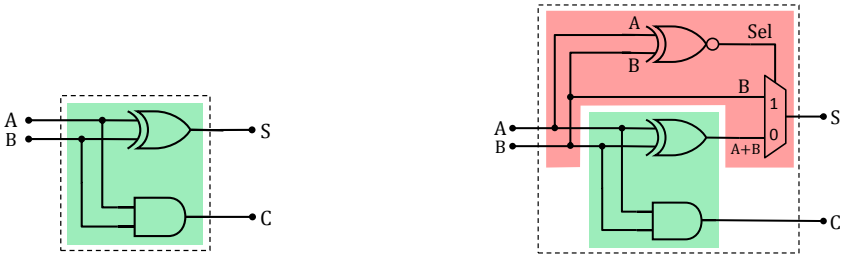
3.1 IP Core Design Space

An IP core can fall under one of the following three categories, as shown in Figure 1, based on its level of conformity to the design specifications; (1) containing *A Hardware Trojan*, or (2) *An Exploit*, or (3) exhibiting *Normal Behavior*. We define *extra* circuitry as redundant logic added to the IP core without which the core can still meet its design specifications.

3.1.1 Hardware Trojan

A hardware trojan is a malicious logic embedded inside a larger circuit resulting in data leakage or harm to the normal functionality of the circuit once activated. A *Trigger Activated* trojan is one which activates upon some special internal or external event termed as *Trigger*. Whereas an *Always Active* trojan is one which remains active all the time to deliver the payload. Once activated, a trojan can deliver its payload either through standard I/O channels or through side channels (also called covert channels). Denial of service, performance degradation, privacy leakage, reducing the reliability of the device, weakening of the security mechanisms e.g. bypassing the side channel protection circuitry, discarding counter measures etc. are some examples of possible payloads of a hardware trojan.

Figure 2 shows an example of a simple hardware trojan embedded in a half adder circuit. The trojan free circuit is shown in Figure 2a which, under normal behavior, generates a sum $S = A \oplus B$ and a carry $C = A \cdot B$. The trojan circuitry, highlighted in red in Figure 2b, triggers when $A = B$. Note that this trigger condition becomes less probable if A and B are several bits wide vectors. The payload of this trojan is to generate incorrect results i.e. $S = B$ for $A = B$ and $S = A \oplus B$ for $A \neq B$. Once the trigger condition occurs, the select line Sel of the multiplexer becomes 1 and the output S is produced by taking a different *branch* through the multiplexer. This is a very



(a) Half Adder

(b) Half Adder with Hardware Trojan: $S = B$ when $A = B$

Figure 2: Example of a simple Hardware Trojan: Figure 2a shows the circuit without the hardware trojan. Figure 2b shows the circuit with the hardware trojan where the trojan circuitry is highlighted in red. The trojan gets activated when $A = B$ and produces incorrect output S.

naïve example of a hardware trojan as it is easily detectable through functional testing (i.e. if $S = A \oplus B$ and $C = A \cdot B$) since its trigger condition is not rare. However this example gives the general intuition about complex and sophisticated trojans which also get triggered from branches due to rarely occurring events. Notice the property that there exists a wire (the select line *Sel* in this case) in the circuit that only toggles when the trojan gets triggered and the circuit manifests malicious behavior. We will define $H_{t,1}$ to be the class of hardware trojans with the property that there exists a wire that only toggles when manifesting malicious behavior. We formally define malicious behavior and hardware trojan classes based on the number of trigger wires in section 5 and 6 respectively.

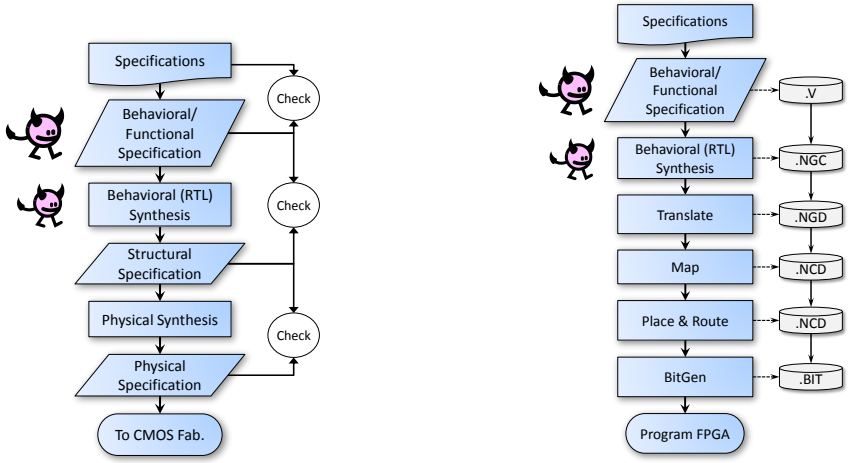
3.1.2 An Exploit

An exploit refers to a loophole in the specifications or implementation of an IP core which allows an adversary to manipulate it beyond its intended specifications. Notice that an exploitable IP core does not have an extra circuitry like a hardware trojan. Depending upon the nature of the exploits, they can also deliver the same payloads as hardware trojans such as privacy leakage or denial of service etc. Some examples of exploits are as follows; a wireless connection that can be overloaded may lead to denial of service, a broken AES module due to a predictable key, OpenSSL Heartbleed bug etc.

This paper focusses on providing a formal framework for rigorous reasoning about hardware trojans (not exploits).

3.2 IP Core Design Flow & Adversarial Model

Digital ASIC design process starting from the high level design specifications to the final chip fabrication involves several design phases as shown in Figure 3a. First, high level design specifications are captured and modeled using some system specification language e.g. C, C++, SystemC or SystemVerilog. This system level design is transformed to RTL design where the system level specifications are modeled using a hardware description language (HDL) such as Verilog or VHDL. After verifying the functionality through simulations, RTL synthesis is performed which generates a generic gate level description, usually called *synthesized netlist*, from RTL description and also



(a) ASIC Design Flow

(b) FPGA Design Flow

Figure 3: Digital IP Core Design Flow: Since a user is provided the IP core after RTL synthesis phase, therefore in our model only RTL synthesis and earlier design phases as considered vulnerable to inclusion of a hardware trojan and hence marked as malicious.

performs logic optimization for speed and area. The synthesized netlist is passed through further design phases (e.g. physical synthesis, floorplanning, place and route etc.) and eventually transformed into a chip.

The IP core vendors provide their IP cores typically after the RTL synthesis phase which obfuscates the HDL source code. Therefore in our model we only consider RTL synthesis and earlier design phases as vulnerable to inclusion of a hardware trojan. Rest of the design phases are considered trusted since the user is in control of the additional added logic around the IP core (if any) to produce a larger design.

Notice that up to the RTL synthesis phase, the design flow for FPGAs is also pretty similar to that of ASICs as shown in Figure 3b. The only major difference is that for FPGAs, the synthesis process maps the design to the device specific components such as LUTs, flipflops and DSP units etc. rather than the generic logic gates. Therefore our framework applies to the FPGAs as well, although the computational complexity is increased due to the limitation of device specific components in synthesized netlist for FPGAs.

4 Classification of Hardware Trojans

Based on their activation mechanism and payload delivery channels, hardware trojans can be broadly classified into four types as shown in Figure 4. *TA*-Xx and *AA*-Xx refer to Trigger Activated and Always Active trojans respectively. XX-*St* and XX-*Si* refer to the trojans using standard I/O channels and side/covert channels respectively to deliver the payload. I/O channels are generally used to communicate binary payloads b_j at certain times t_j for the duration of the execution of the IP core. In this sense the view of an I/O channel can be represented as a sequence $(b_1, t_1), (b_2, t_2), \ldots, (b_N, t_N)$. Its information is decomposed in three channels: the binary channel corresponding to $(b_1, b_2, \ldots, b_N)$, the timing channel corresponding to $(t_1, t_2, \ldots, t_N)$, and the ter-

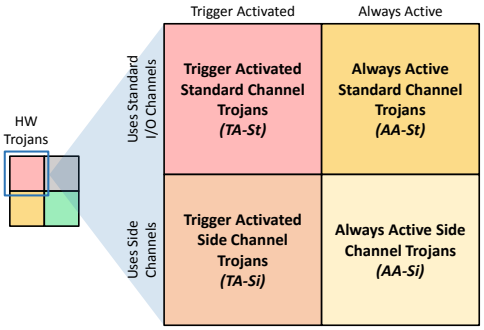


Figure 4: Classification of Hardware Trojans from Figure 1 based on their activation mechanism and payload delivery channels.

mination channel N which reveals information about the duration of the execution of the IP core. If a trojan delivers some of its payload over the timing covert channel (or other side channels), then we define it to be in XX-*Si*. If a trojan delivers *all* of its payload using the standard usage of I/O channels (the binary and termination channels), then we define it to be in XX-*St*. E.g., a hardware trojan causing performance degradation in terms of slower response/termination times due to slower computation (denial of service in the most extreme case) is in XX-*St*.

In Figure 5, we further refine our description of *XX-St* trojans by distinguishing those trojans that are embedded in an IP core which output is a function of its input, i.e. the logical functionality of the IP core is deterministic, and those trojans whose malicious payload changes the logical functionality of the IP core. We denote these properties by D and F respectively, and ND and NF denote their negations. E.g., a trojan which degrades performance in that it slows down the response times over the standard I/O channels up to the worst-case acceptable by the logical functional specification of (user interaction with) the IP core, does therefore not violate the specified logical functionality and is in class *XX-St-X-NF*. If performance is degraded to the extreme of a denial of service, then logical functionality does change since an answer is expected within a reasonable time frame. Such trojans as well as trojans that change logical functionality in general are in *XX-St-X-F*. The reason for differentiating deterministic versus non-deterministic IP cores is that we can show that HaTCh guarantees a zero-false negative rate for *XX-St-D-F* trojans which we cannot expect to achieve for trojans that are not *XX-St-D-F*. We note however that *any* trojan that uses triggering mechanisms as employed by *XX-St-D-F* trojans will be detected by HaTCh. Our evaluation section considers all known benchmarked trojans and most of them happen to have this property. This shows the relevance of creating a formal framework around *XX-St-D-F* trojans only in section 5.

5 Formal Hardware Trojan Framework

In order to model and define hardware trojans, we will first provide a relaxed model for the input and output behavior of IP cores in sections 5.1 and 5.2. We will explain how *XX-Si* trojans can abuse the probabilistic nature of side channels to remain undetected. Similarly, *XX-St-ND-X* trojans, which use standard I/O channels to covertly embed malicious payload, remain undetected by exploiting any non-deterministic behavior of the IP core. Finally, we notice that a *XX-St-X-NF* trojans cannot be detected using a logical specification alone. For the remaining class of *XX-St-D-F*

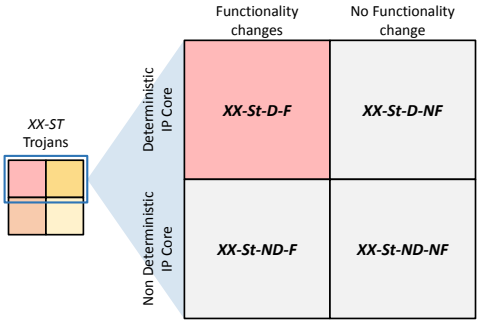


Figure 5: Classification of *XX-St* type Hardware Trojans from Figure 4 based on the behavior of the IP core and the trojan.

trojans we will set up a rigorous framework in this section and section 6.

5.1 IP Core

An IP core *'Core'* written in a higher language (such as Verilog, VHDL) represents a circuit module $M = M^{Core}$ (with feed-back loops, internal registers with dynamically evolving content, etc.) that receives inputs (over a set of input wires) and produces outputs (over a set of output wires). We define the *state* of M at a specific moment in time (measured in cycles) as the vector of binary values on each wire inside M together with the values stored in each register/flip-flop. Here, the definition of state goes beyond just the values stored in the registers inside M: M itself may not even have registers that store state, M's state is a snapshot in time of M's combinatorial logic (which evolves over time). By S_i we denote M's state at clock cycle i.

To complete the picture, we define x_i as the input crafted by a user of the IP core which is received by M over its input wires at clock cycle i (this excludes default input wire values during clock cycles when input is not given by the user, i.e. input wires are undriven). Similarly, we define y_i as the output of the IP core which is destined for the user and is generated by M over its output wires at clock cycle i. We write $x_i = \epsilon$ or $y_i = \epsilon$ if no input is given or output is produced. Triples (x_i, y_i, S_i) model the evolving state of M assuming an influx of inputs x_i coming from a user:

1. The IP Core starts in an initial state S_0.
2. During the i-th clock cycle, $i \geq 1$, the IP core generates on input x_i

$$(y_i, S_i) \longleftarrow M^{Core}(x_i, S_{i-1}).$$

In a first attempt, we model the user as a probabilistic polynomial time (ppt) algorithm $User$ which, based on previously generated inputs and received outputs, constructs new input that is received by the IP core in the form of a new value x_i:

$$(x_{i+1}, U_{i+1}) \leftarrow User(y_i, U_i),$$

where U_i indicates the current state of the algorithm (which is derived from the history of input and output pairs (x_j, y_j), $j \leq i$) and where algorithm $User$ computes (based on a newly received output y_i) a new state U_{i+1} and a new input x_{i+1} for the IP core. In this model the user interacts

with the IP core from cycle to cycle. A malicious user in this model is therefore able to monitor precise timing information: it is in his interest to embed a hardware trojan that makes optimal use of this timing (side) channel by sometimes delaying an output for just a single cycle in order to covertly transmit private information to the malicious user. Clearly, an ordinary user does not have such fine-grained interaction and cannot (and does not want to have to) verify precise timing specifications of a fine-grained functional specification of the IP core. For this reason we will relax the user interaction with the IP core leading to a more coarse-grained functional specification where not individual transmitted bits per clock cycle are considered but only semantical units of transmitted information over longer periods of time.

We assume (malicious) users who, due to (network) latencies, cannot observe detailed timing information. Even stronger, we assume an adversarial model in which the attacker cannot observe any side channels such as the timing channel, power channel, heat map, etc. I.e., in our analysis we exclude *XX-Si* trojans. Due to the probabilistic nature of side channels, a *XX-Si* trojan can always embed some minimal malicious payload without being detected by an external observer who considers it normal to see small probabilistic fluctuations (allowing a trojan to embed information which looks like acceptable probabilistic fluctuations over the side channel). The probabilistic nature of side channels prohibits the development of a tool which recognizes malicious behavior based on a functional specification to detect *XX-Si* trojans with zero false negatives. The presence of a non-zero false negative rate in an adversarial model that allows *XX-Si* trojans implies a constant rate of privacy leakage.[2] It is outside the scope of this paper to analyze side-channel models/frameworks in existing literature that may lead to tools that can detect *XX-Si* trojans with small false negative rates or obfuscate (by adding extra circuitry) the effect of *XX-Si* trojans leading to reduced privacy leakage rates.

In this paper we assume adversaries who do not have access to side channels. I.e., no physical access to the power pins, or having the possibility to capture a heat map etc.; and only (remote) access to the I/O pins such that (network) latencies obfuscate the timing channel. See algorithm 1, we model this by restricting $User$ to a ppt algorithm with two alternating modes; an *input generating mode* and a *listening mode*. During the input generating mode, some input message X_j is generated which is translated to a sequence $(x_k, x_{k+1}, \ldots, x_n)$ of input vectors for each clock cycle to the circuit module M which defines the IP core. Whereas in the listening mode, which is Δ_j clock cycles long, $User$ collects an output message Y_j that efficiently represents the sequence of output vectors $(y_g, y_{g+1}, \ldots, y_k, y_{k+1}, \ldots, y_n, \ldots, y_{n+\Delta_j})$ as generated by M during clock cycles from the end of the last input generating mode at clock cycle g onwards, i.e., the output generated during clock cycles $g, g+1, \ldots, n+\Delta$. In other words, *User* simply produces an input message X_j, waits to receive an output message Y_j, produces a new input message X_{j+1} etc. (the duration of the input generating and listening modes may vary over time):

$$(\Delta_{j+1}, X_{j+1}, U_{j+1}) \leftarrow User(Y_j, U_j),$$

where U_0 denotes a fixed initial state of $User$. I.e., the view of an (adversarial) user is the sequence $(\Delta_1, X_1, Y_1, \Delta_2, X_2, Y_2, \ldots)$, where X_{j+1} depends on X_h and Y_h, $1 \leq h \leq j$, and Y_j depends on X_h and Y_{h-1}, $1 \leq h \leq j$ (with $Y_0 = \epsilon$). The X_j are produced as semantic units of input that arrive over several clock cycles at the IP core. Y_j concatenates all the meaningful ($\neq \epsilon$) outputs that were generated by the IP core since the transmission of X_j. This means that the view of the user is

[2]Therefore, as a design principle, IP cores that need to protect against (maliciously engineered) leakage over side channels must implement leakage resilient key renewal, e.g., as described in [31].

simply an ordered sequence of values devoid of any fine grained clock cycle information. Therefore, in this model only *XX-St* trojans (which make use of standard I/O channels) can be used in an effective way by an adversary.

Algorithm 1 Algorithmic description of the interaction between M^{Core} and $User$.

procedure INTERACTION
  $g, Y_0, j = 1, \epsilon, 1$
  while true **do**
    $(\Delta_j, X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$
    $(x_k, \ldots, x_{k+n}) \leftarrow \text{SEND}(X_j)$
    $(x_g, \ldots, x_{k-1}) = (\epsilon, \ldots, \epsilon)$
    $(x_{k+n+1}, \ldots, x_{k+n+\Delta_j}) = (\epsilon, \ldots, \epsilon)$
    for $i \leftarrow g, k+n+\Delta_j$ **do**
      $(y_i, S_i) \leftarrow M^{Core}(x_i, S_{i-1})$
      if $y_i \neq \epsilon$ **then** $Y_j = Y_j || y_i$
      end if
    end for
    $j, g = j+1, k+n+\Delta_j+1$
  end while
end procedure

5.2 Specification

In the interaction depicted in algorithm 1 an "ideal" user can verify the correctness of the received output Y with respect to its input X according to a functional specification. Here we assume that the listening mode lasts for a sufficient number of clock cycles Δ such that a semantic unit of output can be received in time. Note that Δ can be considered as part of the specifications that takes into account worst-case transmission latencies to the user, the pipeline depth of the core, etc.

In order for an ideal user to be able to verify functional correctness we assume that the IP core has an algorithmic functional specification consisting of two algorithms: *CoreSim* and *OutSpec*.

5.2.1 CoreSim

An algorithm that simulates the IP core at the coarse grain level of semantic output and input units:

- *CoreSim* starts in an initial state S'_0
- $(Y'_j, S'_j, \Delta_j) \longleftarrow CoreSim(X_j, S'_{j-1})$

CoreSim should be such that it does not reveal any information about how the IP core implements its functionality. It protects the intellectual property (implementation and algorithmic tricks etc.) of the IP core and only provides a specification of its functional behavior. States S'_j are not related to the states S_i that are snapshots of the circuit module M as represented by *Core*. States S'_j represent the working memory of the algorithm *CoreSim*.

Notice that $CoreSim$ also outputs Δ_j, the listening time needed in algorithm 1 to receive Y_j if a user would interact with M^{Core} instead of $CoreSim$. Since $CoreSim$ is public knowledge, $User$ can execute $CoreSim$ as a subroutine to compute its Δ_j values.

5.2.2 OutSpec

Specifies which standard output channels should be used and how they should be used. Standard output channels are defined as those which can be configured by the hardware itself (by programming reserved registers etc.). E.g., a hardware trojan doubling the Baud rate (by overwriting the register that defines the UART channel) or a hardware trojan which unexpectedly uses the LED channel (by overwriting the register that programs LEDs), as implemented in [32], would violate $OutSpec$. As discussed previously, side channel attacks are defined as attacks which use non-standard output channels (which are not covered by $OutSpec$).

At this point, couple of remarks are in place:

First, an "ideal" user who can verify the correctness of received output may not exist since the initial state S'_0 may not be known because it should encode private information of the IP core, e.g., a private key which ought not be exposed outside the IP core. In general, a user may become in possession of an IP core which already evolved towards an unknown state S_i. In our model, however, we assume that IP cores can be (reset to a known default state S_0 and) programmed (through the input channel) with an initial state. This implies that a user of the IP core can (after a reset) know the initial state S_0 and its corresponding state S'_0 in $CoreSim$, and can be considered an "ideal" user.

Second, we assume that the language (e.g. Verilog, VHDL) used by the vendor to describe the provided IP core netlist $Core$ allows the user of the IP core to emulate its fine grained behavior, i.e., we assume an algorithm $Emulate$:

- $Emulate[Core]$ starts in an initial state S_0.
- $(y_i, S_i) \longleftarrow Emulate[Core](x_i, S_{i-1})$.

$Emulate[Core]$ behaves exactly as the circuit module M corresponding to $Core$, i.e. $Emulate[Core]$ and M are functionally the same. The main difference is that $Emulate[Core]$ parses the language in which $Core$ is written: If programmed in software, this leads to a huge performance overhead. If implemented in hardware, $Emulate[Core]$ needs to explicitly output the states S_i and this means that extra circuitry (logic and registers) needs to be added to M (to some extent the extra circuitry can be thought of as a standard testing circuit). Notice that $Emulate[Core]$ is solely based on $Core$ and cannot leak more information about the intellectually property of the IP core than described by $Core$ itself.

In practice, one can think of $Emulate[Core]$ as any post-synthesis simulation tool, such as *ModelSim*, which can be used to simulate the provided IP core netlist $Core$. Notice the following properties of such a simulator tool; firstly it does not leak any information about the IP other than described by $Core$ itself and secondly, it is inefficient in terms of (completion time) performance since it performs software based simulation, however it provides fine grained information to the user about the internal state of the IP core at every clock cycle.

To summarize the discussion, the user of the IP core is in a unique position to use $Emulate[Core]$ and verify whether its I/O behavior (over standard channels) matches the specification $(CoreSim, OutSpec)$. The verification can be done automatically without human interaction: This

Algorithm 2 $User$ interacts with $Emulate[Core]$ for J rounds and verifies functional correctness and outputs the list of all the emulated states of M^{Core}.

procedure SIMULATE($Core, User, J$)
  $g, Y_0, j, States = 1, \epsilon, 1, []$
  while $j \leq J$ **do** ▷ J rounds
    $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$ ▷ No output Δ_j
    $(Y'_j, S'_j, \Delta_j) \leftarrow CoreSim(X_j, S'_{j-1})$ ▷ Spec.
    $(x_k, \ldots, x_{k+n}) \leftarrow$ SEND(X_j)
    $(x_g, \ldots, x_{k-1}) = (\epsilon, \ldots, \epsilon)$
    $(x_{k+n+1}, \ldots, x_{k+n+\Delta_j}) = (\epsilon, \ldots, \epsilon)$
    for $i \leftarrow g, k+n+\Delta_j$ **do** ▷ Emulate
      $(y_i, S_i) \leftarrow Emulate[Core](x_i, S_{i-1})$
      if $y_i \neq \epsilon$ **then** $Y_j = Y_j || y_i$
      end if
      $Append(States, S_i)$ ▷ Update $States$
    end for
    $j, g = j+1, k+n+\Delta_j+1$
    if $Y'_j \neq Y_j$ **then** ▷ Verification
      return ("Trojan-Detected", $States$)
    end if
  end while
  return ("OK", $States$) ▷ All emulated states
end procedure

leads to our HaTCh tool which uses (during a *learning phase*) $Emulate[Core]$ to simulate the actual IP core M^{Core} from algorithm 1 and which verifies whether the sequence $(X_1, Y_1, X_2, Y_2, \ldots)$ computed by algorithm 1 matches the output sequence $(Y'_1, Y'_2, \ldots)$ of $CoreSim$ on input $(X_1, X_2, \ldots)$, where we assume $User$ to output the same listening times Δ_j as $CoreSim$. Algorithm 2 shows a detailed description of this process. After this simulation based verification, HaTCh will create (during a *tagging phase*) additional circuitry which will 1) test the register settings of the standard output channels as defined in $OutSpec$ and 2) flag unexpected behavior which HaTCh had not seen in one of the states S_i in $States$ during the execution of algorithm 2. The detailed methodology of HaTCh is explained in section 7.

5.3 *XX-St* Trojan Behavior

We first observe that $CoreSim$ may not be able to exactly match the output (as in $Y'_j = Y_j$ in algorithm 2) of a non-deterministic M^{Core}, which generates and uses true random bits. Since $CoreSim$ cannot generate (pseudo) random bits that are the same as the true random bits in M^{Core}, the output of $CoreSim$ may slightly differ from that of M^{Core}. In this case verification is based on some similarity measure. As explained in section 5.1 for the timing channel, since verification based on a similarity measure allows small probabilistic fluctuations in the output of M^{Core}, a hardware trojan can be designed that uses this artifact to leak privacy at a non-zero rate. We will therefore not be able to offer strong security guarantees for our HaTCh tool in case of non-deterministic IP cores. Even though HaTCh can be applied to non-deterministic IP cores, we will assume in the remainder of this paper deterministic IP cores, i.e., $CoreSim$ is a non-probabilistic algorithm. This means that the output sequence $(Y'_1, Y'_2, \ldots)$ of $CoreSim$ is uniquely defined (and next definitions make sense).

We define the input sequence $X_1, X_2, \ldots, X_N$ to elicit *normal behavior* if it verifies properly in algorithm 2, i.e., if the emulated output (by $Emulate[Core]$) correctly corresponds to the simulated output (by $CoreSim$). An input sequence $X_1, X_2, \ldots, X_N$ manifests *malicious behavior* if it does not elicit normal behavior.

6 Hardware Trojan Class $H_{t,d}$

A *XX-St-D-F* trojan that manifests malicious behavior must have transitioned through a state of "no return" (also called a trigger state) after which malicious behavior manifests itself in the form of a payload that violates the functional specification (*XX-St-D-F* trojans are defined to only exhibit this kind of malicious behavior) *regardless of the other subsequent user interactions.* Clearly, since the number of possible states is finite, there must be some upper bound t on the number of cycles within which malicious behavior manifests after a state of "no return". We (informally) define H_t as the set of *XX-St-D-F* trojans that meet upper bound t. A formal definition of H_t is as follows:

Definition 1. *$Core \in H_t$ if and only if the following conditions hold:*

C1) There exists a $User$ and number of rounds J such that SIMULATE*$(Core, User, J)$ outputs "Trojan-Detected". I.e., $Core$ is indeed capable of malicious behavior in the form of a violation of the functional spec.*

C2) For all $User$ and number of rounds J with the property that SIMULATE*$(Core, User, J)$ returns ("Trojan-Detected", $States$), there exists a state $S = States[j]$ for some $|States| - t \leq j \leq$*

$|States|$ *such that for all* $User'$, J', *and* j'

$$\begin{array}{l} \left[\begin{array}{l} (., States') \leftarrow \textsc{Simulate}(Core, User', J') \\ S = States'[j'] \end{array}\right] \\ \Longrightarrow \\ |States'| \leq j' + t \end{array}$$

Here, S represents a trigger state after which, independent of any future interaction by another $User'$, either the IP core aborts within t clock cycles without eliciting malicious behavior or malicious behavior is manifested within t clock cycles.

In order to understand how this relates to algorithm 2, we define the concept of t-legitimate states.

Definition 2. *We define the set of t-legitimate states of $Core$ as*

$$L_t^{Core} = \left\{ S : \begin{array}{l} \exists \\ {\scriptstyle User, J} \end{array} \begin{array}{l} \exists \\ {\scriptstyle 1 \leq j < |States| - t} \end{array} \text{ such that } \begin{array}{l} (., States) \leftarrow \textsc{Simulate}(Core, User, J) \\ S = States[j] \end{array} \right\}$$

These states can be reached by M^{Core} up to t clock cycles before normal behavior ends or malicious behavior starts manifesting.

If a hardware trojan has the property that it always manifests malicious behavior within t clock cycles after "it gets triggered", then we know that the states in L_t do not show the "trigger signal". This means that we can use the states in L_t to *whitelist* certain wires or combinations of wires from the IP core. The following lemma formalizes this insight and simplifies H_t's definition.

Lemma 1. *$Core \in H_t$ if and only if*

$\mathcal{C}1^*$) ($\mathcal{C}1$) holds.

$\mathcal{C}2^*$) For all $User$ and number of rounds J with the property that $\textsc{Simulate}(Core, User, J)$ outputs ("Trojan-Detected", States), there exists an index $|States| - t \leq j \leq |States|$ such that state $States[j] \notin L_t$.

Proof. Suppose ($\mathcal{C}2$) holds. If $S \in L_t$, then by its definition there exists a $User'$ and number of rounds J' such that $\textsc{Simulate}(Core, User', J')$ outputs $States'$ with $S = States'[j']$ for some $1 \leq j' < |States'| - t$. From ($\mathcal{C}2$) we infer that $|States'| \leq j' + t$, a contradiction, hence, $S \notin L_t$ implying ($\mathcal{C}2^*$).

Suppose ($\mathcal{C}2^*$) holds. Suppose the negation of ($\mathcal{C}2$), i.e., there exists a $User'$, J', and j' such that $\textsc{Simulate}(Core, User', J')$ outputs $States'$ with $S = States'[j']$ for some $1 \leq j' < |States'| - t$. Then $S \in L_t$ contradicting ($\mathcal{C}2^*$). □

Since hardware trojans may have trigger signals that involve a small number, say d, wires, it is useful to define projections:

Definition 3 (Projections)**.** *We define a vector* $\mathbf{z}$ *projected to index set* P *as* $\mathbf{z}|P = (z_{i_1}, z_{i_2}, \ldots, z_{i_d})$ *where* $P = \{i_1, i_2, \ldots, i_d\}$ *and* $i_1 < i_2 < \cdots < i_d$*. We call* d *the dimension of projection* P *and we define* $\mathcal{P}_d$ *to be the set of all projections of dimension* d*. We define a set* Z *projected to index set* P *as* $Z|P = \{\mathbf{z}|P : \mathbf{z} \in Z\}$

Now we define the whitelist $W_{t,d}$ and its corresponding blacklist $B_{t,d}$. We remind the reader that S represents a state *vector*, not a set.

Definition 4 ($W_{t,d}$ and $B_{t,d}$)**.**

$$W_{t,d} = \{(P, S|P) \ : \ P \in \mathcal{P}_d \ , \ S \in L_t\}$$
$$B_{t,d} = (\mathcal{P}_d \times \{0,1\}^d) \setminus W_{t,d}$$

We observe that $S \notin L_t$ is equivalent to

$$\exists_d \exists_{P \in \mathcal{P}_d} \ S|P \notin L_t|P$$

I.e. $S \notin L_t$ implies $S|P \notin L_t$ for the identity projection which leaves states unchanged; if $S|P \notin L_t|P$ then $S \notin L_t$ by the definition of projection $L_t|P$. For this reason it makes sense to define the subclass $H_{t,d}$.

Definition 5. *$H_{t,d}$ consists of $Core \in H_t$ such that for all $User$ and number of rounds J with the property that* SIMULATE*$(Core, User, J)$ outputs ("Trojan-Detected", States), there exists an index $|States| - t \leq j \leq |States|$ and a projection $P \in \mathcal{P}_d$ such that $S|P \notin L_t|P$. I.e., $(P, S|P) \notin W_{t,d}$ or equivalently $(P, S|P) \in B_{t,d}$.*

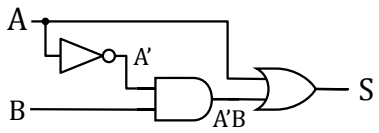
If $Core \in H_{t,d}$, then trigger states S manifest themselves in d of its wires in a way that cannot be observed for normal behavior. Notice that $H_{t,1} \subseteq H_{t,2} \subseteq \ldots \subseteq H_t$.

6.1 $H_{0,1}$ Class Trojan Example

In section 3, we have shown an example of a simple $H_{0,1}$ hardware trojan in Figure 2b. In this circuitry, one can distinguish between the normal behavior and a trigger event by observing the logic values of each wire individually. This is possible since at least one wire in the circuit (in this case the select line Sel) does not flip during the normal operation. Specifically Sel wire remains 0 during the normal operation. As this trojan can be distinguished by observing 1-wire combinations it has dimension $d = 1$ and since the trojan exhibits its malicious behavior immediately after the trigger event occurs it has $t = 0$. Hence it is a $H_{0,1}$ trojan.

6.2 $H_{0,2}$ Class Trojan Example

Figure 6 shows another implementation of the same trojan, having the trigger condition $\{A, B\} = \{1, 1\}$, from the previous example. Here, the circuitry from Figure 2b is optimized in a way that trojan circuitry is merged into the normal circuitry. Therefore all the internal wires are part of producing the normal results too as it is evident from the table in Figure 6b. Hence one cannot distinguish the trigger condition by observing individual wires or 1-wire combinations. However, certain 2-wire combinations in this design only occur upon the trigger condition, for example $\{A, B\} = \{1, 1\}$, $\{B, A'\} = \{1, 0\}$ and $\{B, A'B\} = \{1, 0\}$. Since one can only distinguish between



(a) $S = A \oplus B$ affected to produce $S = A + A'B$

A	B	A'	$A'B$	S
0	0	1	0	0
0	1	1	1	1
1	0	0	0	1
1	**1**	**0**	**0**	**1**

(b) Truth Table of the trojan affected circuit. Trigger: $\{A, B\} = \{1, 1\}$

Figure 6: A $H_{0,2}$ trojan embedded in summation function of a Half Adder: Figure 6a shows the trojan from Figure 2b implemented in a way that the trojan circuitry cannot be distinguished from the normal circuitry, i.e. there is no single wire which only toggles under trigger condition. This can be seen in the truth table of Figure 6b.

the normal behavior and the trigger event of this trojan by observing 2-wire combinations, and it shows malicious behavior immediately after the trigger condition therefore it is a $H_{0,2}$ trojan.

For completeness, we mention that DeTrust in their recent paper (section 3.3 in [10]) created a trojan design which turns out to be in $H_{1,2}$ according to our hierarchical model.

We also show another example of a counter based $H_{0,2}$ trojan in Appendix B.

6.3 $H_{0,d}$ Class Trojan Example

Figure 7 depicts *k-XOR-LFSR*, a counter based trojan with the counter implemented as an LFSR (with a primitive feedback polynomial) of size k. Let $r^i \in \{0,1\}^k$ denote its register content at clock cycle i represented as a binary vector of length k. Suppose that u is the maximum index for which the linear space L generated by vectors $r^0, \ldots, r^{u-1}$ (modulo 2) has dimension $k-1$. Since $dim(L) = k-1 < k = dim(\{0,1\}^k)$, there exists a vector $v \in \{0,1\}^k$ such that (1) the inner products $\langle v, r^i \rangle = 0$ (modulo 2) for all $0 \leq i \leq u-1$ and (2) $\langle v, r^u \rangle = 1$ (modulo 2). Only the register cells corresponding to $v_j = 1$ are being XORed with inputs A_j. The expected hamming weight of v is $k/2$, which is the number of XORs of register cells with the A_j as depicted in figure 7.

Since the A_j are all XORed together in the specified logical functionality to produce the sum $\sum_j A_j$, the trojan changes this sum to

$$\sum_j A_j \oplus \sum_{j:v_j=1} r_j^i = \sum_j A_j \oplus \langle v, r^i \rangle.$$

I.e., the sum remains unchanged until the u-th clock cycle when it is maliciously inverted. For the proof of the following lemma, see appendix A.1.

Lemma 2. *Suppose that all vectors r^i behave like random vectors from a uniform distribution. Then k-XOR-LFSR has register size k and triggers after $u \approx k$ clock cycles. Furthermore, k-XOR-LFSR is $\notin H_{0,\approx \log k - 2\log\log k}$ but is in $\in H_{0,d}$ for some larger d.*

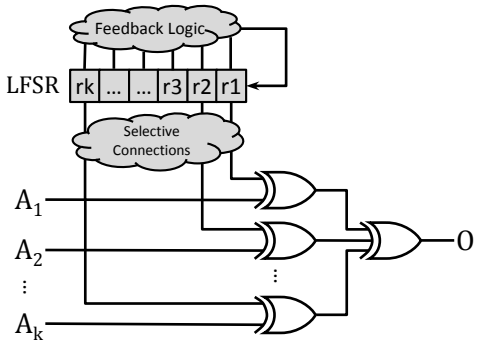


Figure 7: *k-XOR-LFSR*: A general $H_{t,d}$ hardware trojan.

7 HaTCh Framework

We propose a guaranteed hardware trojan disabler tool called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh). The key idea which makes the HaTCh tool really powerful is based on *whitelisting*. In general, a whitelist refers to a list of those entities which are provided a particular privilege, service or recognition. Whitelists limit the trusted base to a finite number of trustworthy entities. HaTCh adapts the same approach to discriminate the trustworthy circuitry of the IP core from its potentially malicious parts.[3]

Algorithm 3 shows the operation of HaTCh. In order to disable any $H_{t,d}$ type hardware trojan in $Core$, HaTCh processes $Core$ in two phases; a *Learning phase* and a *Tagging phase*. The learning phase puts $Core$ through functional testing and returns a blacklist B of unused wire combinations as explained in section 6. Notice that this blacklist is different from a conventional blacklist which is created by blacklisting the untrusted entities. This blacklist is created by taking the complement of the corresponding whitelist W and therefore it contains 'everything' other than the trusted entities (wire combinations in this case), which eliminates the possibility of a false negative. If no malicious behavior is observed during the learning phase, then the tagging phase starts. It transforms $Core$ to $Core_{Protected}$ by adding extra logic for each entry in the blacklist such that whenever any of these wires is activated, a special warning signal will be asserted to indicate the activation of a potential hardware trojan. Otherwise if $Core$ is found manifesting any malicious behavior during the learning phase then the learning phase is immediately terminated. This produces an error condition and as a result, HaTCh does not execute its tagging phase and simply returns "Trojan-Detected" which indicates that the IP core contains a hardware trojan and is rejected straightaway.

7.1 Learning Phase

HaTCh$(Core, t, d)$ attempts to learn the blacklist $B_{t,d}$ of $Core$. Algorithm 4 describes this learning phase in which we iterate over H users, each describing a different way of interacting with the IP core. E.g., in the case of an AES core in encryption mode, which only receives one input X_1 (the plain text) and produces one output Y_1 (the cipher text) per user interaction, each $User_h$ can be thought of as an algorithm which hard-codes its own plain text input X_1. We iterate over many

[3]The English word *Hatch* means an opening of restricted size allowing for passage from one area to another. Our HaTCh framework provides this functionality by allowing certain parts of the circuit to operate normally while restricting the operation of others.

Algorithm 3 HaTCh Algorithm

```
1: procedure HaTCh(Core, t, d)
2:     (w, B) ← Learn(Core, t, d)
3:     if w ≠ "Trojan-Detected" then
4:         Core_Protected ← Tag(Core, B)
5:         return Core_Protected                    ▷ The Protected Core
6:     else
7:         return "Trojan-Detected"
8:     end if
9: end procedure
```

such users to learn the AES core's behavior for many different inputs. In Learn( ) we assume that Simulate( ) starts $Emulate[Core]$ in a state S corresponding to where the last application of Simulate( ) left off, i.e., Simulate( ) "continues" simulating new users (without resetting the IP core).

Learn( ) simulates one user (one plaintext input in the case of an AES core) after another. This means that the next user (plaintext input) can only be simulated once the output of the previous user (the ciphertext output corresponding to the previous plaintext) has been received. The current description of Learn( ) does not allow a pipelined implementation/simulation where multiple user interactions are handled together by the IP core. E.g., a pipelined AES core may take 10 cycles to produce a ciphertext but does so by computing 10 ciphertexts in parallel giving a throughput of 1 ciphertext per cycle. Without getting into details, we will assume a PipelinedSimulate( ) in Learn( ) that has a stream of $User_h$ as input with their interactions pipelined and interleaved together.

In algorithm 4,

$$L = \cup_{h=1}^{H} \cup_{i=1}^{|States_h|-t-1} \{States_h[i]\} \subseteq L_t$$

by the definition of L_t. Blacklist B is constructed as the complement $B = (\mathcal{P}_d \times \{0,1\}^d) \setminus W$ of W representing the whitelist

$$W = \cup_{S \in L} \cup_{P \in \mathcal{P}_d} \{(P, S|P)\}$$

Combining both equations proves that $W \subseteq W_{t,d}$ by the definition of whitelist $W_{t,d}$, hence:

Lemma 3. *If* Learn*$(Core, t, d)$ outputs a blacklist B, then*

$$B_{t,d} \subseteq B \subseteq \mathcal{P}_d \times \{0,1\}^d$$

7.1.1 Zero False Negatives

Notice that $B_{t,d}$ contains two types of wires (or wire combinations for $d > 1$); first the wires specifically related to the hardware trojan circuitry, and second some redundant wires which did not excite during the learning phase either because of insufficient user interactions or because of logical constraints of the design. This has two consequences: first, since HaTCh will tag all its learned blacklisted wire combinations in B in a subsequent *tagging phase* (see section 7.2), unnecessary tagging circuitry for detecting redundantly blacklisted wire combinations will cause unnecessary area overhead (we will discuss optimizations that deal with this effect). Second, since HaTCh uses Learn$(Core, t, d)$ to learn a superset B of $B_{t,d}$:

Algorithm 4 Learning Scheme

```
1: procedure LEARN(Core, t, d)
2:     if I/O register does not match OutSpec then
3:         return "Trojan-Detected"
4:     else
5:         B ← P_d × {0,1}^d
6:         Get users (User_h, J_h), 1 ≤ h ≤ H
7:         for h ← 1, H do
8:             (w, States_h) ← SIMULATE(Core, User_h, J)
9:             if w = "Trojan-Detected" then
10:                return "Trojan-Detected"
11:            else
12:                for all P ∈ P_d do
13:                    for all 1 ≤ i ≤ |States_h| − t − 1 do
14:                        B = B \ {(P, States_h[i]|P)}
15:                    end for
16:                end for
17:            end if
18:        end for
19:        return B                                   ▷ The Blacklist
20:    end if
21: end procedure
```

Theorem 1. HATCH*(Core, t, d) (its detailed description is given in section 7) will detect all* $Core \in H_{t,d}$ *with zero false negatives.*

7.1.2 Non-Zero False Positives

Since HaTCh learns a superset of $B_{t,d}$, HaTCh will have a non-zero false positive rate:

Definition 6. *The tagging circuitry on top of* M^{Core} *produced by* HATCH*(Core, t, d) causes a* false positive *for User if it incorrectly claims that* $Core \in H_{t,d}$ *at some moment during the execution of the tagged* M^{Core} *when it interacts with User.*

The benchmark IP cores used in the evaluation (section 8) always have a single input X_1 and output Y_1 per user interaction (we stress that our model is more general and extends to IP cores which expect multiple interactions with each $User$). Therefore, for the benchmarks, HaTCh iterates during the learning phase over H users $User_h$ each of them generating only one input "$X_{1,h}$". The probability that $X_{1,h}$ gives rise to a false positive detection by the tagged circuitry is what is defined above. In the evaluation section we show that the false positive rate, measured/estimated by one over the number of user instantiations of the core during which the black list B is not further reduced (hence, a tagged circuitry corresponding to B would not generate false or true positives during these user instantiations), can be made as small as $1/10^5$ depending on the duration of HaTCh's learning phase, measured as the number H of simulated user interactions $User_h$.

We notice that, interestingly, false positives related to the normal functioning of circuitry that has not yet passed through a "point of no return" trigger state can in fact prevent trojan cir-

cuitry to evolve its state to a point of no return. This demonstrates that the blacklist B learned by HaTCh$(Core, t, d)$ potentially detects trojans outside $H_{t,d}$ in H_t as a false positive. E.g., the *k-XOR-LFSR* trojan in $H_{0,\geq \log k - 2\log\log k}$ of section 6.3 needs a learning phase up to clock cycle $u \approx k$ (when it triggers) in order to have the blacklist sufficiently reduced; if the learning phase is up to $\sqrt{k}$, then we need to replace u by $\sqrt{k}$ in formula (2) of appendix A showing that HaTCh$(Core, 0, (\log k)/2 - 2\log\log k)$ likely detects the trojan as a false positive.

We can prove the following theorem about false positives:

Theorem 2. *Suppose there exists a distribution $\mathcal{U}$ with the following properties:*

$\mathcal{U}1$) If $(User, J) \leftarrow \mathcal{U}$, then Simulate*$(Core, User, J)$ outputs States with $c \leq |States| \leq C$ (i.e. each simulation is at least for c clock cycles and can last as long as C clock cycles).*

$\mathcal{U}2$) For all d-dimensional projections P, distribution $\mathcal{U}$ induces a distribution on $States[i]|P$ which is uniform *over $L_t|P \subseteq \{0,1\}^d$.*

Suppose HaTCh*$(Core, t, d)$ uses a learning phase* Learn*$(Core, t, d)$ such that the H users $(User_h, J_h)$ in* Learn*$(\,)$ are drawn from $\mathcal{U}$. Then the mean time between consecutive false positives among a series of users $(User'_h, J'_h) \leftarrow \mathcal{U}$ that each interact with M^{Core} is at least T if*

$$H \approx (2^d/c) \cdot \ln(|Core|^d CT),$$

where $|Core|$ denotes the size of its state vectors (i.e., number of encoded wires).

For the evaluated benchmarks $c = C = 1$ (corresponding to a throughput of 1 output and 1 input per cycle for pipelined simulation), $d = 1$ (all benchmarks are in $H_{t,1}$), $|Core| \approx 10^4$, and $T \approx 10^5$, the learning phase should simulate about $H \approx 42$ users. Clearly, in reality the uniform distribution assumption is not satisfied, therefore we expect the need for a learning phase with a factor more users (a uniform distribution is a best case assumption for white listing and, hence, reducing the number of false positives). Our evaluations show the need for $\approx 10^4$ users (each being thought of as an input in one pipelined simulation) in order to achieve a mean time of $T = 10^5$ between false positives, a factor 250× more. Nevertheless the theorem is useful in that it shows that increasing d exponentially increases the complexity of the learning phase (as measured in $H \cdot |Core|$) of HaTCh.

Proof. Let $P \in \mathcal{P}_d$. Notice that Learn$(Core, t, d)$ simulates in total $\geq Hc$ clock cycles leading to at least Hc wire combinations $S|P$ that are white listed. The probability that a certain wire combination $S|P$ is blacklisted by Learn$(Core, t, d)$ is therefore at most $(1 - 1/f_P)^{Hc}$, where $f_P = |(L_t|P)|$. The probability that in a subsequent interaction of a $User'$ with M^{Core} the wire combination $S|P$ is encountered during one of C clock cycles (leading to a false positive) is equal to $1 - (1 - 1/f_P)^C$. Summing over all P and possible wire combinations in $L_t|P$ gives an upper bound on the probability of $User'$ causing a false positive:

$$\sum_{P \in \mathcal{P}_d} f_P (1 - 1/f_P)^{Hc} (1 - (1 - 1/f_P)^C)$$

By using the inequalities $(1 - (1 - 1/f_P)^C) \leq C/f_P$ and $1 - 1/f_P \leq 1 - 2^{-d}$ we obtain the upper bound

$$\sum_{P \in \mathcal{P}_d} C(1 - 2^{-d})^{Hc} = |\mathcal{P}_d| C (1 - 2^{-d})^{Hc}$$

By noticing that $|\mathcal{P}_d| = \binom{|Core|}{d} \leq |Core|^d$ and $(1-2^{-d})^{Hc} \leq e^{-x}$ for $Hc = x2^d$ we arrive at the upper bound

$$|Core|^d C e^{-Hc/2^d}$$

which is at most $1/T$ if $H \approx (2^d/c) \cdot \ln(|Core|^d CT)$ □

In order to reduce the complexity of the learning phase, HaTCh may use specific knowledge about the type of trojans it wants to detect. E.g., appendix A.2 proves that the *k-XOR-LFSR* trojan of section 6.3 can be detected by a modified HaTCh*$(Core, t)$ which learns all linear invariants (vectors whose inner product with states in L_t is always 0):

Lemma 4. HaTCh*$(Core, 0)$ *creates tagging circuitry which detects k-XOR-LFSR with zero false negatives and zero false positives. Its learning phase has complexity* $O(|Core|^3)$.

7.2 Tagging Phase

The tagging phase implements precautionary measures against the potentially malicious circuitry of the IP core identified in the learning phase. It takes an untrusted $Core$ along with a blacklist B and adds additional logic to the $Core$ to keep track of these suspicious wires. This process is explained in algorithm 5. A new output signal called $TrojanDetected$ is added to the $Core$. This output is asserted whenever any wire from B takes a 'blacklisted' value. To achieve this functionality, a tree of logic gates is added to $Core$ such that the logic 1 is propagated to $TrojanDetected$ output whenever a 'blacklisted' value is taken by a suspicious wire. All gates in the logic tree are 4-input gates (i.e. a quad tree) hence there are $\log_4(|B| \cdot d)$ levels in the quad tree where $|B|$ shows the number of entries in the blacklist B and d shows the dimension of the trojan. Note that this quad tree consists of only combinational logic, i.e. non-pipelined tagging circuitry.

Algorithm 5 Tagging Scheme

```
1: procedure TAG(Core, B)
2:     Add TrojanDetected output port to Core
3:     for all (P, S|P) ∈ B do
4:         TrojanDetected = 1
5:     end for
6:     TrojanDetected = 0 otherwise
7:     return Core                                  ▷ The protected Core
8: end procedure
```

7.2.1 Pipelined Logic

Since the added quad tree has $\log_4(|B| \cdot d)$ combinational logic levels, it could end up on the critical path and hurt the timing of the IP core for large B. To avoid this problem, the tagging phase can also add pipeline registers at every level of the tree which leads to a pipelined tagging circuitry. The pipeline resisters may delay the detection of hardware trojan by $log_4(|B| \cdot d)$ cycles. However we show in our evaluation section that for average sized IP cores, HaTCh produces a significantly small B in reasonable amount of computational time. Consequently, the detection delay because of pipeline registers is also not too large and may still be acceptable. Additionally, for a particular

IP core the HaTCh computation needs to be done only once for millions of its instances to be fabricated. Hence, even for larger IP cores, it is worth investing a computational time of several hours to achieve a significantly small blacklist B and to produce millions of trustworthy chips.

7.2.2 Area Overhead

We discussed false negatives and false positives in section 7.1. Another metric of crucial importance is the area overhead caused by the additional tagging circuitry. The added quad tree has $|B| \cdot d$ leafs. Therefore the number of its internal nodes N_{int} is given by $(\frac{4|B| \cdot d - 1}{3} - |B| \cdot d)$. For a non-pipelined tree, the required number of gates is equal to N_{int}. For a pipelined tree, considering that the outputs of all the added gates except the root node are sampled in a pipeline register, the number of required registers is equal to $N_{int} - 1$. Putting everything together, the non-pipelined area overhead O_{nP} and pipelined area overhead O_P are given by:

$$\begin{aligned} O_{nP} &= (N_{int}) \text{ 4-input gates} \\ O_P &= (N_{int}) \text{ 4-input gates} + (N_{int} - 1) \text{ flipflops} \end{aligned} \tag{1}$$

Theorem 3. *Suppose* HaTCh$(Core, t, d)$ *tags all wire combinations in a blacklist* B*, then the tagging circuitry has an area overhead given by* (1)*.*

This result shows that it is important to research optimization techniques that can reduce $|B|$ by eliminating the redundant blacklisted wire combinations that can never occur (e.g., the input wires and output wire of a simple NAND gate have wire combinations that can never occur given its truth table). We discuss such optimizations implemented by HaTCh in section 8.

7.3 HaTCh Process Example

In order to further clarify the operation of HaTCh, we show an example of how HaTCh works on a real circuit. Figure 8 shows a generic IP core with a counter based trojan. The trojan consists of a trigger circuitry and a payload circuitry and gets triggered when the counter is full, i.e. $Counter = 255$. Once this IP core is passed through the *learning phase* of HaTCh such that the range $0 \leq Counter \leq 127$ is covered, all other wires of the trigger circuitry except $b1, b2, b3$ and $b4$ are whitelisted. These four wires remain stuck at 0 during the learning phase and hence get blacklisted. The blacklisted wires result in added circuitry 'HaTCh Tagging Circuitry' by the *tagging phase* which generates the $TrojanDetected$ signal. Notice that in this example, 'HaTCh Tagging Circuitry' is a pipelined tree which is based on the raw blacklist, i.e. without performing any optimizations on it which are discussed in the next section.

8 Evaluation

In this section, we present the experimental results of certain hardware trojan benchmark groups from Trusthub [1] benchmark suite evaluated under HaTCh. We first analyze the Trusthub benchmarks w.r.t. the hardware trojan classification introduced in section 3.1. Then we briefly describe our experimental setup and HaTCh methodology from a more practical standpoint, including some crucial optimizations implemented to reduce the number of the blacklisted wires. Next we present and discuss the experimental results and finally we propose some other optimizations in order to reduce HaTCh complexity and its area overhead.

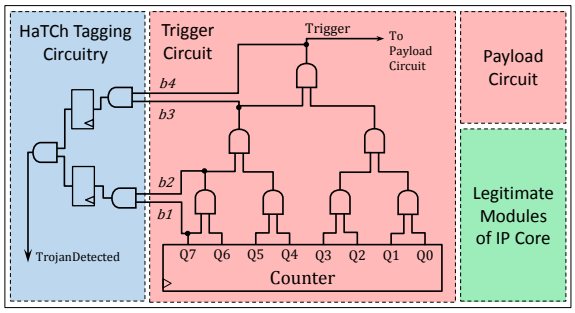


Figure 8: Example of HaTCh operation on a real circuit: A counter based trojan trigger has a trigger condition $Counter = 255$. If HaTCh *learning phase* runs from $Counter = 0$ to 127 then only $Q7$ and its dependent wires are blacklisted and corresponding tagging circuitry is added accordingly.

8.1 Trusthub Benchmark Suite

In order to provide concrete guarantees about which trojans can be detected by HaTCh under all circumstances and which ones can be detected under certain conditions, we categorize the relevant benchmarks from Trusthub [1] benchmark suite in Table 1 according to our formal hardware trojans classification presented in section 4.

8.1.1 XX-St-D-F Trojans

HaTCh detects all *XX-St-D-F* trojans with 100% guarantee. This includes *TA-St-D-F* and *AA-St-D-F* trojans. As explained in earlier sections, the main focus of HaTCh is *TA-St-D-F* trojans, i.e. trigger activated trojans which manifest malicious behavior in terms of change in normal functionality of deterministic IP cores. However it also detects *AA-St-D-F* trojans always in the *learning phase* because such trojans are meant to always exhibit malicious behavior over the standard I/O channels which can be easily detected by functional testing during the learning phase. Notice that *AA-St-D-F* is a weak class of trojans and it is practically not useful for an adversary to design such trojans as they have no stealthiness.

8.1.2 TA-Si and TA-St-D-NF Trojans

HaTCh detects all *TA-Si* and *TA-St-D-NF* trojans as well provided that these trojans do not get triggered during the HaTCh learning phase. *TA-Si* trojans exploit side channels to deliver the malicious payload and do not affect the normal functionality over standard I/O channels. Therefore during the learning phase, even if such a trojan gets activated, HaTCh will not observe any malicious behavior and whitelist the trigger related wires. On the other hand, if it does not get activated during the learning phase, only the non-malicious wires will be whitelisted and the trigger related wires would still remain in the blacklist leading to trojan detection. The same reasoning is applicable to *TA-St-D-NF* trojans.

Table 1: Classification of Trusthub Benchmarks w.r.t. HaTCh framework

Classification			***Benchmarks***
XX-St	***TA-St***	***TA-St-D-F***	BasicRSA-T {100, 200, 300, 400} EthernetMAC10GE-T700-T{700, 710, 720, 730} MC8051-T{200, 300, 400, 500, 600, 700, 800} PIC16F84-T{100, 200, 300, 400} vga-lcd-T100 wb_conmax-T{100, 200, 300} b15-T{100, 200, 300, 400} b19-T{100, 200, 300, 400, 500} s15850-T100 s35932-T{100,200,300} s38417-T{100, 200} s38584-T{100, 200, 300} RS232-T{100, 300, 400, 500, 600, 700, 800, 900, 901, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1900, 2000}
		TA-St-D-NF	AES-T{500, 1800, 1900} RS232-T{200, 1800}
	AA-St	***AA-St-D-F***	N/A
		AA-St-D-NF	EthernetMAC10GE-T{100, 200, 300, 400, 500, 600} MultPyramid-T100-T{100, 200}
XX-Si	***TA-Si***		AES-T{400, 600, 700,800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 2000, 2100} s38417-T300
	AA-Si		AES-T{100, 200, 300}

8.1.3 AA-Si and AA-St-D-NF Trojans

These types of trojans are out of the scope of HaTCh and cannot be detected. Since these trojans are not trigger-based, therefore they are always active during the learning phase. However such trojans always show malicious behavior in a way that does not harm the normal functionality of the IP core and hence HaTCh considers the circuitry as 'normal' and whitelists it.

We evaluate HaTCh using three benchmark groups, s15850, s35932 and s38417. In the rest of this section, we refer to these benchmarks as *s-Series* benchmarks. These three groups contain seven different benchmarks in total which are of different nature based on their activation mechanism and payload channels. However, all these benchmarks are at $H_{t,1}$ level in our hierarchical model. Notice that s38417-T300 belongs to *TA-Si* but since it does not get triggered in the learning phase, HaTCh is still able to detect it.

8.2 Experimental Setup & Methodology

8.2.1 RTL Synthesis

HaTCh works on a gate level netlist with a flat hierarchy. Therefore first the RTL design needs to be synthesized. We use the Synopsys logic synthesis tool *Design Compiler* [33] for this purpose. However the s-Series benchmarks are already provided as synthesized netlists therefore we skip this step.

8.2.2 Simulation

Next step is to perform post-synthesis simulations using self checking testbenches for which we use Mentor Graphic's *ModelSim* [34] simulator. The benchmarks are given random test patterns as inputs and the output is compared with the expected output in the self-checking testbench. If the output is different than expected, the simulation is aborted indicating the activation of a hardware trojan.

Obviously, Automatic Test Pattern Generator (ATPG) tools are very popular to generate the test patterns which provide significantly higher coverage compared to random test patterns. However ATPG tools do not fit the HaTCh settings because the internal state of the design reached by an ATPG test pattern may not be reachable by the normal inputs given at the standard input ports. Therefore, the self checking testbench will not be able to compute an expected output for this ATPG test pattern to compare with the output generated by the design. Consequently, the simulator will not be able to distinguish a 'normal' output from a malicious output to abort the simulation if a trojan gets triggered by an ATPG test pattern during the simulation.

The documentation of s-Series benchmarks does not provide any information about the normal functionality of these cores therefore we cannot compute an expected output to compare with the generated output in the self-checking testbenches. As a workaround to this problem, we use a trick in our experiments. Since the trojan design and documentation is available to us, we know which wire is the trigger signal. We design the testbenches which monitor this signal during the simulation, and the simulation is aborted if the trigger signal is activated. If not, the simulation finishes 'normally' and the whole simulation data is dumped into a VCD file which contains the simulation trace of each wire.

8.2.3 HaTCh Learning and Tagging Phases

In the next phase a HaTCh-script parses the VCD file, generated during the simulation phase, in order to learn and blacklist the unused wires of the circuit. Initially all the wires of the circuit are put in the blacklist. Then the transitions of each wire is read by the script, and an internal data structure is updated which eventually leads to a blacklist of wires (or combinations of wires for $d > 1$) which do not show any transition. Based on the blacklist, additional logic is added to flag the blacklisted wires.

8.2.4 HaTCh Optimizations

The blacklist generated in the learning phase contains three different types of wires. **First**, those which constitute a true blacklist and are part of the hardware trojan trigger circuitry. These wires, once activated, are going to trigger the hardware trojan. **Second**, the wires which are in the

blacklist because of the low coverage of the input test patterns. **Third**, the redundant wires in the blacklist which are not necessary to guarantee 100% detection rate. For example if the input(s) and the output of certain logic elements (gates, buffers) exist in the blacklist at the same time, then it is sufficient to keep only the input in the blacklist provided that changing this input will affect the output, e.g. in case of logic buffers and inverters etc. Redundancy is also caused by certain wire combinations which are logically not possible, e.g. for $d = 2$, the combination of the input and output wire of an inverter is never going to have a value of 00 or 11. The first type of wires generates *true positives* whereas second and third types of wires generate *false positives* and lead to unnecessary area overhead.

The third type of wires, however, can be avoided by careful optimizations based on the logical constraints of the design. Our HaTCh tool performs these optimizations in order to remove the redundant wires form the blacklist. The key idea behind these optimizations is that if the input(s) and output of a logic element coexist in the blacklist, then the output wire can be removed from the blacklist provided that changing this input will affect the output. HaTCh performs these optimizations, wherever possible, for all buffers, logic gates and flipflops in the design. Our experiments show that these optimizations lead to a significant reduction in the size of blacklist which in turn reduces the area overhead.

8.3 Experimental Results

8.3.1 s-Series Benchmarks

We ran HaTCh on seven different benchmarks from three different groups of s-Series, i.e. s15850-T100, s35932-{T100, T200, T300} and s38417-{T100, T200, T300} and HaTCh detected all of them. The trojan in one of the benchmarks, i.e. s35932-T100 was detected already during the learning phase as it got triggered and caused the simulation to be aborted. For the rest of the benchmarks, we ran up to 2×10^5 input test patterns. Figure 9 shows the size of blacklist sampled at seven different numbers of input patterns. The size of the blacklist decreases rapidly with the number of input patterns until it reaches a state when most of the wires in the design are already whitelisted and no more wires are eliminated from the blacklist by further testing. It can be seen that the sizes for the s35932 group already become stable after 100 input patterns. Whereas the s38417 group achieves the stable state after 10,000 input patterns. Only the s15850 group takes longer to become stable where only one wire is reduced between 100,000 and 200,000 input patterns.

This leads to an important result about the false positives of HaTCh. Since only one wire is eliminated from the blacklist while going from 100,000 to 200,000 input patterns, therefore it would cause only one false positive for the next 10^5 inputs if the blacklist obtained by running 100,000 inputs is used to generate the tagging circuitry. Hence the false positive rate is $1/10^5$.

8.3.2 Other benchmarks

We have analyzed the source code/netlists of other *TA-St-D-F* benchmarks from Table 1 and we have concluded that all these benchmarks reside in $H_{t,1}$ in our hierarchical model and therefore HaTCh will detect all of them.

In Appendix B, we also show how HaTCh can detect $H_{0,2}$ trojans given the parameter $d = 2$.

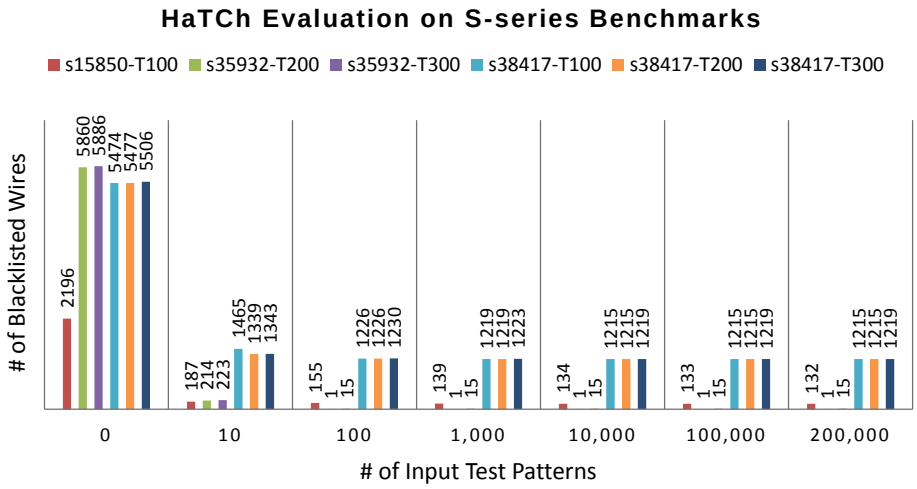


Figure 9: The experimental results for s-Series benchmarks: the absolute size of the blacklist for different number of test patterns.

Table 2: Area Overhead for s-Series Benchmarks

Benchmark	***Size***	***Area Overhead***			
		Pipelined		***Non-Pipelined***	
		Un-Opt	***Opt***	***Un-Opt***	***Opt***
s15850-T100	2180	33.44%	4.17%	16.74%	2.11%
s35932-T200	5442	9.83%	0.02%	4.92%	0.02%
s35932-T300	5460	10.27%	0.16%	5.15%	0.09%
s38417-T100	5341	57.27%	15.22%	28.65%	7.62%
s38417-T200	5344	57.24%	15.21%	28.63%	7.62%
s38417-T300	5372	57.41%	15.25%	28.70%	7.63%
Average		37.58%	8.34%	18.80%	4.18%

8.4 Area Overhead

Table 2 shows the area overhead incurred by HaTCh for each of s-Series benchmarks both for non-pipelined and pipelined tagging circuitry. The overheads caused by the raw unoptimized blacklists obtained after the learning phase are shown under *Un-Opt* whereas the overheads after performing optimizations introduced in section 8.2.4 are shown under *Opt*. The size of benchmarks in terms of total number of gates and registers is shown under *Size*. On average, we see an overhead of 38% and 19% for unoptimized pipelined and non-pipelined circuitries respectively. The overhead drops down to 8.34% and 4.18% for optimized pipelined and non-pipelined circuitries respectively. This shows the significant impact of the optimizations performed by HaTCh which reduce the overhead by ≈ 4.5 times.

8.5 Techniques to reduce HaTCh Complexity

In order to further reduce the computational complexity of HaTCh, we propose the following techniques:

8.5.1 Golden Input Patterns

Since the high level RTL design of the core is not available to us, the test patterns we use for simulations are randomly generated which may not provide good coverage. In order to achieve a good coverage and a higher efficiency during the learning phase, the manufacturer could provide the golden input patterns, which are designed based on the high level RTL, to the user or verifier keeping the high level RTL design secret.

8.5.2 Modular Approach for IP cores

Unsurprisingly, a larger design needs huge number of test patterns to excite all the paths in the circuit. These large designs are typically composed of small modules, where some of the modules maybe instantiated several times. Therefore, verifying these small modules independently rather than a large design composed of these modules would be much more efficient. Also, after verification, the users must integrate these small modules by themselves or be able to verify the integration process done by untrusted parties. Notice that this technique assumes that all the trigger related wires of the trojan are contained within one module, i.e. the trigger circuit does not consist of cross-modular wires.

8.6 Techniques to reduce HaTCh Area Overhead

8.6.1 Formal Proofs to reduce the blacklist

As explained earlier, some of the wires in the blacklist are the result of poor coverage of functional test patterns and may not be relevant to the hardware trojan in any way, but since they remain unused during the simulations they are blacklisted. By using formal methods, if one can prove that these wires will either never flip in future or if they do, they are not going to exhibit malicious behavior, then these wires can be removed from the blacklist.

8.6.2 Clustering approach for high level HaTCh

Another approach to reduce the blacklist size and the computational complexity for higher level HaTCh (i.e. for $d > 1$) is to only investigating the combinations of wires with other wires within a predefined maximum distance. This technique relies on the fact that the trigger related wires of a practical high dimension hardware will be in the close vicinity of each other due to the limited hardware overhead budget and the complexity involved due to the wire delays with higher distance.

9 Conclusion

We have provided a hierarchical model for a certain class of trojans with increasing complexity. We have introduced HaTCh, a powerful hardware detection tool, which detects all existing trojans belonging to a certain class of hardware trojans that change logical functionality based on a trigger

signal with zero false negatives, significantly low false positives and low area overhead. Finally we have proposed techniques for future work in order to reduce the computational complexity and area overhead of HaTCh.

References

[1] M. Tehranipoor, R. Karri, F. Koushanfar, and M. Potkonjak, "Trusthub," http://trusthub.org.

[2] V. Tomashau and T. Kean, "Validation of an advanced encryption standard (aes) ip core," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 291–292. [Online]. Available: http://dl.acm.org/citation.cfm?id=1025123.1025845

[3] S. Morioka and A. Satoh, "A 10 gbps full-aes crypto design with a twisted-bdd s-box architecture," in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 98–103.

[4] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 2013, pp. 40–49.

[5] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Theory of Cryptography*. Springer, 2014, pp. 1–25.

[6] S. Bhasin, J.-L. Danger, S. Guilley, X. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic ip cores," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, Aug 2013, pp. 15–29.

[7] M. Hicks, M. Finnicum, S. King, M. Martin, and J. Smith, "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.

[8] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "Veritrust: Verification for hardware trust," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 61:1–61:8. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488808

[9] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 697–708. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516654

[10] 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 & Communications Security*, 2014, *to appear*.

[11] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *Design Test of Computers, IEEE*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[12] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, ser. LEET'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 5:1–5:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387709.1387714

[13] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, no. 5, pp. 34–39, May 2008.

[14] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 23–40.

[15] Y. Liu, Y. Jin, and Y. Makris, "Hardware trojans in wireless cryptographic ics: Silicon demonstration & detection method evaluation," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 399–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561908

[16] T. Reece, D. Limbrick, X. Wang, B. Kiddie, and W. Robinson, "Stealth assessment of hardware trojans in a microcontroller," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, Sept 2012, pp. 139–142.

[17] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Provably complete hardware trojan detection using test point insertion," in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, Nov 2012, pp. 569–576.

[18] J. Zhang and Q. Xu, "On hardware trojan design and implementation at register-transfer level," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, June 2013, pp. 107–112.

[19] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating uci: Building stealthy and malicious hardware," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 64–77. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.32

[20] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 49–63. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.27

[21] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, Oct 2010.

[22] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Hardware trojan horse benchmark via optimal creation and placement of malicious circuitry," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 90–95. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228378

[23] R. Chakraborty and S. Bhunia, "Security against hardware trojan through a novel application of design obfuscation," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, Nov 2009, pp. 113–116.

[24] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 396–410.

[25] X. Wang, H. Salmani, M. Tehranipoor, and J. Plusquellic, "Hardware trojan detection and isolation using current integration and localized current analysis," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on*, Oct 2008, pp. 87–95.

[26] S. Narasimhan, X. Wang, D. Du, R. Chakraborty, and S. Bhunia, "Tesr: A robust temporal self-referencing approach for hardware trojan detection," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, June 2011, pp. 71–74.

[27] K. Hu, A. N. Nowroz, S. Reda, and F. Koushanfar, "High-sensitivity hardware trojan detection using multimodal characterization," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1271–1276.

[28] S. Narasimhan, D. Du, R. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia, "Hardware trojan detection by multiple-parameter side-channel analysis," *Computers, IEEE Transactions on*, vol. 62, no. 11, pp. 2183–2195, Nov 2013.

[29] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware trojans," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 632–639. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509456.1509596

[30] D. Forte, C. Bao, and A. Srivastava, "Temperature tracking: An innovative run-time approach for hardware trojan detection," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, Nov 2013, pp. 532–539.

[31] C. Hazay, A. López-Alt, H. Wee, and D. Wichs, "Leakage-resilient cryptography from minimal assumptions." in *EUROCRYPT*. Springer, 2013, pp. 160–176.

[32] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 2009, pp. 50–57.

[33] "Synopsys Inc." http://www.synopsys.com, Mountain View, CA.

[34] "ModelSim, Mentor Graphics Inc." www.mentor.com , http://www.model.com, Wilsonville, OR.

A Analysis of *k-XOR-LFSR*

A.1 Classification

The *k-XOR-LFSR* trojan uses an LFSR to generate register values $r^i \in \{0,1\}^k$ for each clock cycle i. Suppose that all vectors r^i behave like random vectors from a uniform distribution. Then, (1)

it is unlikely that u is more than a small constant larger than k (since every new vector r^i has at least probability 1/2 to increase the dimension by one). Therefore, $u \approx k$, hence, the register size of the trojan is comparable to the number of clock cycles before the trojan is triggered to deliver its malicious payload. This makes the trojan somewhat contrived (since it can possibly be detected by its suspiciously large area overhead). (2) By our definition of L_0, all vectors r^i, $0 \leq i \leq u-1$, are part of states in L_0. Consider a projection P to a subset of d register cells. If $r^u|P \in \{r^i|P : 0 \leq i < u\}$, then the wire combination of the d wires corresponding to $r^u|P$ is not black listed: if this is the case for all d dimensional P, then the trojan is $\notin H_{0,d}$. The probability that $r^u|P \in \{r^i|P : 0 \leq i < u\}$ is at least equal to the probability that $\{r^i|P : 0 \leq i < u\} = \{0,1\}^d$, which is (by the union bound)

$$\begin{aligned} &\geq 1 - \sum_{w \in \{0,1\}^d} Prob(\{r^i|P : 0 \leq i < u\} \subseteq \{0,1\}^d \setminus \{w\}) \\ &= 1 - \sum_{w \in \{0,1\}^d} (1 - 1/2^d)^u \approx 1 - 2^d e^{-u/2^d}. \end{aligned}$$

Since there are $\binom{k}{d} \leq k^d/d!$ projections, the trojan is $\notin H_{0,d}$ with probability (taken over all random r^i)

$$\geq (1 - 2^d e^{-u/2^d})^{k^d/d!}. \tag{2}$$

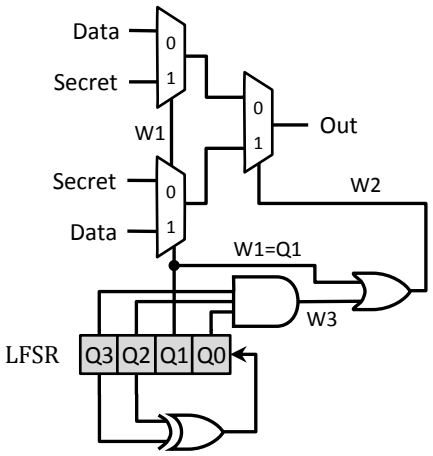
For $u \approx k$, this lower bound is about $\geq 1/e$ for $k^d/d! \leq 2^{-d} e^{k/2^d}$, e.g, $d \approx \log k - 2 \log\log k$. This shows that *k-XOR-LFSR* is $\notin H_{0,\approx \log k - 2\log\log k}$ but is in $\in H_{0,d}$ for some larger d.

A.2 Improving HaTCh

The *k-XOR-LFSR* trojan has the property that $\langle v, r^i \rangle = 0$ for $0 \leq i < u$ until the trojan gets triggered. This implies that, for states S of M^{Core}, there exists a vector V corresponding to v such that $\langle V, S \rangle = 0$ until the trojan is triggered. So, if HaTCh could enumerate all vectors V with $\langle V, S \rangle = 0$ for all states $S \in L_t$ generated during SIMULATE in the learning phase, then a tagging circuitry can be added that checks for these invariants. We notice that these vectors V have the property that they are orthogonal to the linear space $\mathcal{L}_t$ spanned by $L_t \subseteq \{0,1\}^{|Core|}$. It takes $O(|Core|)$ random samples $S \in L_t$ to generate $\mathcal{L}_t$. Once these samples are gathered a simple Gaussian elimination process will discover all vectors V in $O(|Core|^3)$ time. The complexity of the learning phase for this modified HaTCh is independent of d (at the cost of only having zero false negatives for the subset of all H_t trojans with a "linear trigger" signal).

B A Counter-Based $H_{0,2}$ Trojan Example

The example trojan shown in Figure 10a can leak *Secret* via *Out* port instead of *Data* upon the trigger condition $W1 \neq W2$. The trigger condition is generated by a counter, when reached to (1101), which is implemented as a 4-bit maximal LFSR in order to have maximum possible time before the trojan gets triggered. The LFSR is initialized to $(Q3, Q2, Q1, Q0) = (1010)$ and it can be seen in Figure 10b that if given the parameter $d = 1$, HaTCh will whitelist all the wires related to trigger circuitry only after a few clock cycles since all these wires show transitions. At $14th$ clock cycle, the value of the LFSR becomes (1101) and $W1 \neq W2$, which activates the Trojan to leak the secret.



(a) Trojan Circuitry

$Cycle$	$Q3$	$Q2$	$Q1$	$Q0$	$W1$	$W2$	$W3$
0	1	0	1	0	1	1	0
1	0	1	0	1	0	0	0
2	1	0	1	1	1	1	0
3	0	1	1	1	1	1	0
4	1	1	1	1	1	1	1
5	1	1	1	0	1	1	0
6	1	1	0	0	0	0	0
7	1	0	0	0	0	0	0
8	0	0	0	1	0	0	0
9	0	0	1	0	1	1	0
10	0	1	0	0	0	0	0
11	1	0	0	1	0	0	0
12	0	0	1	1	1	1	0
13	0	1	1	0	1	1	0
14	**1**	**1**	**0**	**1**	**0**	**1**	**1**

(b) Truth Table of the Trojan affected circuit. Trigger: $W1 \neq W2$

Figure 10: A Counter-Based $H_{0,2}$ Trojan to enable the secret leakage

B.1 Detection by HaTCh with $d = 2$

As it is clear from Figure 10b that, given the parameter $d = 1$, this trojan cannot by detected by HaTCh since all the wires show transitions and get whitelisted after $4th$ clock cycle. Therefore we run HaTCh with a parameter $d = 2$ in order to show that HaTCh is still able to detect this trojan. With $d = 2$, HaTCh exhaustively monitors all possible 2-wire combinations of all the wires in the design. It starts with a blacklist of all possible 2-wire combinations (e.g. $\{00, 01, 10, 11\}$) of all the wires in the design and those combinations which are seen during the simulation are removed from the blacklist provided that the output Out matches the expected output for every input.

If the learning phase is run for 13 clock cycles, then after the optimizations of HaTCh, we only see one combination of $W1$ and $W3$ in the final blacklist i.e. $(W1, W3) = (0, 1)$ which only occurs upon the trigger condition. All other redundant combinations are optimized away from the blacklist because the logical constraints of the design never allow these combinations to occur in the future, e.g. $(W1, W2) = (1, 0)$ is never possible (unless a stuck at 0 fault for $W2$). Hence HaTCh is able to detect this trojan. Notice that if the learning phase is run for fewer clock cycles, then HaTCh will produce a larger blacklist with more blacklisted combinations.